(4)

Supporting the Transfer of Simulation Technology

James R. Kipps, Iris Kameny,
Jeff Rothenberg

RAND | NATIONAL DEFENSE RESEARCH INSTITUTE

The research described in this report was sponsored by the Defense Advanced Research Projects Agency. The research was conducted in the National Defense Research Institute, RAND's federally funded research and development center supported by the Office of the Secretary of Defense, Contract No. MDA903-85-C-0030.

Library of Congress Cataloging in Publication Data

Kipps, James R. (James Randall), 1960-

Supporting the transfer of simulation technology / James R. Kipps, Iris Kameny, Jeff Rothenberg.

p. cm.

"July 1989."

"Prepared for the Defense Advanced Research Projects Agency."

"R-3740-DARPA."

ISBN 0-8330-0931-1

1. Computer simulation. 2. Programming languages (Computer science) 3. SERAS (Computer system) 4. Technology transfer. I. Kameny, Iris, 1932- . II. Rothenberg, Jeff, 1947- . III. United States. Defense Advanced Research Projects Agency. IV. RAND Corporation. V. Title.

QA76.9.C65K57 1989

003'.3—dc20 89-8442

CIP

Published by The RAND Corporation
1700 Main Street, P.O. Box 2138, Santa Monica, CA 90406-2138

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER R-3740-DARPA	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Supporting the Transfer of Simulation Technology		5. TYPE OF REPORT & PERIOD COVERED interim
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) James R. Kipps, Iris Kameny, Jeff Rothenberg		8. CONTRACT OR GRANT NUMBER(s) MDA903-85-C-0030
9. PERFORMING ORGANIZATION NAME AND ADDRESS The RAND Corporation 1700 Main Street Santa Monica, CA 90406		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Boulevard Arlington, VA 22209		12. REPORT DATE July 1989 13. NUMBER OF PAGES 91
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for Public Release; Distribution Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

No Restrictions

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Simulation
Technology Transfer
Department of Defense

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

see reverse side

DD FORM 1 JAN 73 1473

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

The principal hurdle in transfering new ideas and techniques from the research arena into real-world Department of Defense applications comes from conflicts in the programming language requirements of the two groups involved--simulation researchers and military analysts. This report proposes the development of a common-base programming system for military simulation--SERAS (System for Exploration, Research, and Applications in Simulation). The SERAS programming system is not a new simulation programming language, but a prototyping tool, primarily targeted at simulation researchers. This report summarizes efforts in developing a plan for supporting the transfer of simulation technology and outlines a preliminary design of SERAS. It presents fundamental concepts in simulation and simulation programming; outlines concepts relevant to simulation from artificial intelligence and object-oriented programming; surveys conventional and experimental simulation programming languages; and reviews current perceptions of military simulation at RAND and elsewhere. Finally, the report outlines a preliminary design of the SERAS programming system, and discusses a plan for ~~the~~ development of SERAS. → Technology transfer
Computer and simulation for
Department of Defense
Index.

R-3740-DARPA

Supporting the Transfer of Simulation Technology

James R. Kipps, Iris Kameny,
Jeff Rothenberg

July 1989

Prepared for the
Defense Advanced Research
Projects Agency

RAND

PREFACE

This research was sponsored by the Information Sciences and Technologies Office of the Defense Advanced Research Projects Agency under The RAND Corporation's National Defense Research Institute, a Federally Funded Research and Development Center sponsored by the Office of the Secretary of Defense. Under this sponsorship, RAND, in its Information Processing Systems program, has been investigating the utility of knowledge-based artificial intelligence techniques to help solve deficiencies that exist in large-scale military simulation. The research described in this report is one aspect of this effort.

SUMMARY

Technology transfer is the process by which new ideas and techniques move from the research arena into real-world applications. The objective of the Transfer of Simulation Technology Project is to develop a plan for improving the movement of results from research in advanced simulation programming techniques (i.e., research in knowledge-based and object-oriented simulation) into the RAND simulation laboratories and, eventually, into the Department of Defense (DoD) and services communities.

The principal hurdle in this genre of technology transfer comes from conflicts in the programming language requirements of the two groups involved: simulation researchers and military analysts. Researchers, exploring new approaches to military simulation through knowledge-based and object-oriented programming, require a flexible and dynamic language, such as PROLOG or a LISP-like language, which has built-in support for this style of programming. Analysts, developing and running simulation studies, require a space-efficient language with reliable execution-time performance, such as Fortran or Simscript, which allows them to model complex military systems and quickly run repeated simulation experiments. Unfortunately, to make a language flexible and dynamic, a trade-off between space and execution efficiency is necessary. As a result, languages preferred by researchers seldom coincide with those preferred by analysts.

This situation severely hinders the transfer of advanced technology. Mapping programming techniques developed in a dynamic language (with built-in support for knowledge-based and object-oriented programming) to a static language (without such support) requires more than just a good working knowledge of the languages and the techniques. It is a difficult, time-consuming, and unrewarding task, which neither researchers nor analysts are prepared or willing to undertake. This means that the results of advanced simulation research are seldom available in an efficient, production-level language for analysts to evaluate on realistic models, let alone to use on a regular basis.

Our plan for supporting the transfer of advanced simulation technology is an attempt to narrow this language gap. What we propose is the development of a common-base programming system for military simulation, SERAS (System for Exploration, Research, and Applications in Simulation). The SERAS programming system is not a "new" simulation programming language, but a prototyping tool, primarily targeted at simulation researchers. SERAS is intended to be a vehicle not only

for exploring advanced simulation programming techniques but, more importantly, for incorporating promising techniques into efficient prototype simulation languages. Such prototypes would make these techniques available to military analysts for further testing and evaluation on scaled-up problems. Successful prototypes could later be used as the basis for designing and developing production-level simulation languages.

While SERAS is only a first step toward relieving the bottleneck of technology transfer, it has the potential for significant advantages at RAND and other sites doing related research. It would allow new ideas to be ported from the research environment to an environment that can be used for building realistic simulations, and, thus, have a unifying effect on current research and applications in military simulation.

ACKNOWLEDGMENTS

We would like to thank Stephanie Cammarata and Sanjai Narain for repeatedly reading and critiquing earlier drafts of this report and helping to shape our ideas. Patrick Allen and Al Zobrist provided insightful and cogent reviews. In addition, Patrick Allen and H. E. Hall assisted in patching technical and conceptual gaps in our review of simulation projects and languages. We would also like to thank Judith Westbury for editing this report and Mary Aguilar for her assistance in processing it.

CONTENTS

1. INTRODUCTION

Despite the fact that simulation has a long history of use by the military for training, testing, and analysis, the software technology applied in simulation programming trails the state of the art. A major factor contributing to this situation is the difficulty of technology transfer. Recent research at The RAND Corporation and elsewhere has explored new approaches to simulation programming, borrowing technology from work in artificial intelligence (AI) and object-oriented programming (OOP). While the results of this research have demonstrated the potential effectiveness of these new approaches, they have been very slow to move into regular application. The Transfer of Simulation Technology Project was initiated to address the problem of technology transfer at RAND.

1.1 OBJECTIVE

The objective of the Transfer of Simulation Technology Project is the development of a plan for transferring the results of advanced simulation research into the RAND simulation laboratories and eventually into the Department of Defense (DoD) and services communities. Our approach to technology transfer calls for the further development of a core system for the rapid prototyping of simulation programming languages. The proposed design of this system, called SERAS (System for Exploration, Research, and Applications in Simulation), is dominated by two criteria: (1) that it have the flexibility to be smoothly extended by RAND researchers exploring knowledge-based and object-oriented simulation; and (2) that it have the utility to be applied by military analysts in building realistic models.

The intention behind SERAS is to bridge the gap between the programming environments used by research and application. The immediate purpose of pursuing this approach is not to build a better simulation language; rather, the purpose is to develop a system that aids in porting experimental techniques in computer simulation from a research environment to an applications environment. Such a system will support improvements in simulation programming languages through rapid prototyping, experimentation, and evaluation.

1.2 MOTIVATION

Technology transfer is the process by which ideas and techniques move from the research arena into real-world applications. What makes technology transfer difficult is the time-consuming effort required to make these new ideas *effectively accessible* to the target user group, where "effectively accessible" means that the cost of embracing the new technology does not exceed its potential benefit. In the context of this work, ideas and techniques correspond to knowledge-based and object-oriented techniques for modeling military systems. The cost of such techniques is perceived as reduced execution-time efficiency, limited model size, and an unproductive learning phase.

In recent simulation research, RAND synthesized ideas and approaches from work in AI, OOP, graphics, and distributed computing. Techniques have been developed to make simulations easier to build and maintain, while improving their utility and their comprehensibility. While this work has produced technology to aid in building, maintaining, and utilizing simulations as well as in understanding and presenting their results, this technology has not been adopted by the modeling community at large. This is primarily due to the difficulty of transposing such techniques, developed in dynamic programming environments such as LISP or PROLOG, onto the efficient but static environments, such as Fortran or Simscript, typically used for building military simulations. Such a task requires spanning the worlds of research and application and distilling the new ideas from one into a form that is usable in the other. It is both complex and time-consuming, and there is no guarantee of finishing before new and better technology makes the results of such an effort obsolete.

A naive approach to simplifying technology transfer would require that researchers explore ideas within the environments typically used for simulation, or, conversely, that model builders write simulations in the demonstration environments developed by researchers. Although this approach would narrow the gap between the two groups, it would nonetheless hamper the productivity of one or the other. Each group adopted its respective computing environment because it satisfies essential functionality requirements. Researchers require an environment that enhances their ability to rapidly explore a variety of (often partially defined) ideas, a dynamic, interactive environment that is easily extended and that provides built-in support for advanced programming capabilities. Military analysts require an efficient execution-time environment with the capacity to model large and complex systems. These two basic requirements are inherently at odds with each other; to get a dynamic, extensible environment, one must

sacrifice speed and memory. To force either group to work in the environment of the other would deny that group the basic capabilities they need for functioning in a timely and effective manner.

Despite the difficulties inherent in taking the above approach to its extremes, the notion of having a common base for both research and application has a certain attraction when considering technology transfer. With a common base, techniques developed by simulation researchers could be made available to model builders as extensions to their modeling environment, instead of substitutions. This is the spirit of the approach that we have pursued in the Transfer of Simulation Technology Project.

1.3 APPROACH

Our plan for supporting technology transfer of knowledge-based and object-oriented simulation research is an attempt to narrow the gap between the programming languages used in research and applications. What we propose is the development of a common-base programming system for military simulation, SERAS. The principal objective of this approach is to make advanced simulation programming techniques available for testing and evaluation by military analysts on realistic models.

We use the term *programming system* instead of *programming language* because the design of SERAS combines two application-specific programming languages for the purpose of generating prototype languages for military simulation. No single language can reliably support the requirements of both research and application, but a coordinated system that segregates these activities into separate languages can. In this way, the SERAS programming system is not a "new" simulation programming language, but a prototyping tool.

SERAS is primarily targeted at simulation researchers. It is intended to be a vehicle not only for exploring advanced simulation programming techniques but, more importantly, for incorporating promising techniques into efficient prototype simulation languages. Such prototypes would make these techniques available for further evaluation by military analysts. Successful prototypes could later be used as the basis for designing and developing state-of-the-art production-level simulation languages.

1.4 OVERVIEW OF SERAS

The SERAS programming system consists of two component programming languages: *Core* and *SLAG*. Core is a system development language to be used for defining the semantic base of prototype simulation languages, while SLAG (Simulation LAnguage Generator) is a compiler-generator to be used in defining their surface-level syntax. The product of SLAG is a compiler and interpreter for a target prototype simulation language, generically called SL_i. In combination, Core and SLAG enable the rapid prototyping of advanced simulation languages.

Such an architecture supports technology transfer, because it provides a conduit through which the results of simulation research can be ported, in a straightforward manner, to an applications environment. Given that Core provides built-in support for knowledge-based and object-oriented programming, researchers can use it to develop advanced approaches to simulation programming, incorporate them into the semantic base of a new or existing prototype simulation language, and define an appropriate analyst-oriented syntax for the prototype using SLAG. Given that the execution-time dynamics of Core is sufficiently restricted to allow efficient performance, a prototype simulation language based on Core would then be effectively accessible to military analysts and could be used to demonstrate and evaluate new approaches to simulation programming.

Defining SERAS means defining Core and SLAG. Of the two tasks, the second is the simpler; parsing and compiler technology is relatively well understood and few new problems are introduced here. Core, on the other hand, must support the functionality required to implement the semantics of some arbitrary set of advanced simulation languages. This not only includes the functionality needed to implement knowledge-based and object-oriented simulation techniques, but the functionality to implement features found in conventional simulation languages as well.

The preliminary design we have developed for Core is based upon an informal requirements analysis of conventional simulation technology and advanced simulation research. We reviewed existing simulation projects at RAND and elsewhere, collecting the functionality requirements of these projects and evaluating the languages used in terms of how they support this functionality. In analyzing the requirements of simulation researchers, we reviewed several research projects but primarily focused on the goals and objectives of the Knowledge-Based Simulation Project (KBSim), from which our project was originally spawned.

While SERAS is only a first step toward relieving the bottleneck of technology transfer, it has the potential for significant advantages to RAND and other sites doing related work. The SERAS approach gives researchers a hook into the language of military analysts, allowing the gentle introduction of new technology. This approach also overcomes the problems of applying just a single language to both research and applications, because it separates the language used for defining the semantics of simulation from the language used for encoding simulation programs. SERAS will allow new ideas to be ported from the research environment to an environment that can be used for building realistic models, and, in this way, have a unifying effect on current research and applications in military simulation.

1.5 STRUCTURE OF REPORT

In the remainder of this report, we summarize our efforts in developing a plan for supporting the transfer of simulation technology and outline a preliminary design of SERAS. Sections 2 and 3 present fundamental concepts in simulation and simulation programming. Section 4 outlines concepts relevant to simulation from AI and OOP. Section 5 surveys conventional and experimental simulation programming languages, while Section 6 reviews current perceptions of military simulation at RAND and elsewhere. Finally, Section 7 motivates and outlines a preliminary design of the SERAS programming system, and Section 8 discusses a plan for the development of SERAS.

2. MODELING AND SIMULATION

In this section, we review fundamental aspects of simulation. This discussion is by no means complete. It is intended to introduce terminology and concepts discussed later in this report.

2.1 SIMULATION

Simulation is the manipulation of a model to study the behavior of a system as it operates over time (Kiviat, 1967). The manipulation may be by hand, by a computer, or by a combination of people and computers working together. Simulation allows us to understand how systems work, to determine the factors that influence their behavior, and to observe how they react to environmental changes.

Simulation trades accuracy for convenience. When we use simulation, it is with the implicit understanding that the behavior of a model only approximates the behavior of the actual system. Approximation is acceptable when systems are not accessible to experimentation. For instance, simulation allows us to draw inferences about systems (1) without building them (e.g., if they are only proposed systems), (2) without disturbing them (e.g., if they are operating systems that are costly, time-consuming, or dangerous to study), and (3) without destroying them (e.g., if the object of the study is to test limits of stress).

In this regard, industrial uses of simulation for design, procedural analysis, and performance assessment have been highly rewarding, and a large community of researchers has developed structured techniques and languages for computer simulations. Similarly, simulation is ideally suited to studying military systems. Military applications of computer simulation include planning, prediction, and analysis of combat and peacetime situations; single- and multi-person training and war gaming; and battle-management systems for real-time collation of intelligence data.

2.2 SYSTEMS, THEORIES, AND MODELS

The term *system* refers to a collection of *entities* from a circumscribed sector of reality, such as weapon systems or command units in military operations. Individual entities are characterized by their *attributes* and by *relations* that exist between entities. Entities interact and change via time-consuming processes called *activities*. The *state* of a system is a description of its entities, attributes, relations, and activities at any given time.

A system is simulated by manipulating a *model* of the system as it operates over time. A model is further derived from a *theory* of the system. The key operation of developing a theory is *abstraction*. Abstraction entails eliminating all but the significant attributes and activities of the entities in the system.

The terms theory and model are often used interchangeably. While this does not normally lead to confusion or difficulty, there is an important difference between the two when applied to simulation. *A model is a stylistic interpretation of a theory, formalizing the body of propositions that describe how a system behaves.* Just as there can be many theories of how a particular system behaves, there can be many models that formalize a theory. When building a computer simulation, an analyst first develops a theory of system operations and identifies the entities, activities, and constraints of the system. He then formalizes this theory into a model, and finally implements the model as a computer program.

2.3 SCOPE, GRANULARITY, AND ENVIRONMENT

One of the initial steps in building a simulation is defining the *scope* and *granularity* of the system under study. Scope refers to deciding which entities to include in the system, while granularity refers to fixing the level to which entities are decomposed. For instance, the scope of a theater-level simulation would include all theater forces and commanders, with granularity to the division or battalion level. On the other hand, the scope of a hasty river crossing would include division commanders and their forces, with granularity to the level of platoons or even individual troops.

The scope of a system is determined primarily by the reason for its being identified, isolated, and studied. Entities operate as a system when they are integrated so that the activities of one affect the activities of others. A study of one entity cannot be made in isolation without losing effects caused by this interaction. When problems are viewed in a total system perspective, it becomes clear that properties of

the total system are more important than those of the subsystems operating within it. An example of two subsystems operating together is a supply system and a maintenance system at an Air Force base. One supplies what the other requests and vice versa, so neither can exist without the other. If cost is the metric being studied, then total system cost must be the criterion, not individual maintenance or supply system costs; e.g., reducing stock levels can degrade maintenance performance, while altered maintenance policies can affect demands for replenishment.

Every system resides in an *environment*. The environment of a system represents the external factors that affect it. Activities whose origins and effects remain wholly in the system are called *endogenous*, while those from the environment are called *exogenous*. Exogenous activities can be handled in three ways: (1) the scope of the system can be enlarged to include them; (2) they can be ignored; or (3) they can be treated as inputs to the system. Systems can be classified by the types of activities they contain. Systems with exogenous activities are *open*, while those with endogenous activities only are *closed*. For instance (ignoring gamma particles and magnetic fields), a logic circuit can be viewed as a closed system. On the other hand, the supply/maintenance system mentioned above is open if the demands on maintenance and the availability of supplies are determined by external factors, such as sortie rates and supply lines.

2.4 SYSTEM DYNAMICS

Systems are also characterized by their *static* and *dynamic* structures. The entities that make up a system, their attributes, and the relationships in which they are involved define the system's static structure. The activities in which its entities engage specify its dynamic structure. There can be systems with identical static structure but different dynamic structure and vice versa.

A system is said to be in a certain *state* when its entities have properties unique to that state. Depending on the view taken towards activities, e.g., whether they can interact at discrete points or over periods of time, there can be attributes associated with dynamic as well as static system constructs. For instance, an activity can be fully or partially completed, scheduled, in progress or terminated, waiting for another activity to occur or interrupting another activity, etc. Viewed in a general sense, there is no conflict in describing a state in terms of a static or dynamic phenomenon.

Because simulation is the manipulation of a model to reproduce the behavior of a system as it moves through time, a model used to analyze system dynamics leans heavily on a structure that explains how a system moves ahead from static state to static state. Within such a structure, *time* (simulated time) is typically the major independent variable. Other variables are dependent functions of time. Based on their treatment of dependent variables, simulation models can be classified as either *discrete-event* or *continuous*. Simulations that use both discrete-event and continuous change are termed *combined* simulations.

In discrete-event simulations, dependent variables change (discretely) at specific points in simulation time, referred to as *events*. Time itself may be either continuous or discrete, depending on whether events can occur at any time or only at fixed intervals. On the other hand, in continuous simulations, the value of dependent variables can change continuously over time. As with discrete simulations, time itself may be continuous or discrete, depending on whether the values of dependent variables are available at any point in time or only fixed points.

As an example of discrete and continuous change variables, consider a combined simulation of an aircraft following a fixed-track plan. The aircraft is an entity with four attributes (dependent variables): speed, direction, altitude, and position. The fixed-track plan of the aircraft consists of a sequence of events, each of which causes a discrete change in the speed and direction of the aircraft. The position and altitude of the aircraft are computed continuously, as a function of its speed and direction (as well as its position and altitude at the last change to speed and direction).

2.5 STATISTICAL CONSIDERATIONS

A means of classifying system behavior that has a substantial bearing on the simulation model is based on the certainty of reaching one particular state from another (Fishman and Kiviat, 1967). If a given initial state and sequence of inputs uniquely determine the sequence of states that a system attains, then the system is said to be *deterministic*. On the other hand, if the sequence of states varies randomly, then the system is said to be *stochastic*.

Systems requiring simulation are generally composed of one or more elements that have uncertainty associated with them. The modeling of a stochastic system requires that the variability of the elements of the system be characterized using concepts of probability. Likewise, due to the probabilistic nature of the output from a stochastic simulation, practical analysis of this output requires the use of statistical interpretation.

2.6 COMPONENTS OF A SIMULATION STUDY

A *simulation study* is an attempt to answer a predefined set of questions about a system by means of simulation. Although emphasis is often concentrated on encoding a system model into a running program, this is only one aspect of a simulation study.

A simulation study consists roughly of a six-stage iterative process (Maryanski, 1980):

1. *Statement of Purpose.* The system of interest and the objectives of the study must be formulated. This stage delineates the scope and granularity of the system.
2. *Modeling.* Those aspects of system behavior pertinent to the study must be identified. Given that there can be many models for a system, the exact nature of a system model is determined by the purpose of the study. For example, a system that can have many models (or many theories of how it operates) is an Air Force base. For any particular base, or a generalized base structure, there can be maintenance models, personnel models, operations models, etc. Each model differs in its point of view of the total system, yet each includes the same system elements. Differences appear in the form of varying assumptions about how subsystems interact and by varying degrees of details in specifying system structure.
3. *Programming.* Once a model has been formalized, it must be implemented as a computer program. The selection of the programming language to be used is dependent upon the nature of the model and the resources available.
4. *Experimental Design.* Repeated execution of a simulation program is done to measure the performance of the simulated system on varying inputs. This stage is driven by determining the proper input requirements and the availability of such input.
5. *Validation.* Validation is essentially verifying that the simulation program emulates the behavior of the system. While this is a critical aspect of a simulation study, validation methods vary widely depending on the system being investigated. Validation methods range from carefully checking a program to logically proving its correctness, and from deciding whether its results are reasonable to statistically verifying the expected outcome. Largely, the validation technique employed is determined by the type of information available on the actual system.
6. *Analysis.* A simulation study is expected to answer a set of questions about the system (defined in the statement of purpose). Thus, the results of simulation runs must be analyzed to

understand their meaning. Analysis of stochastic simulations must calculate in the effects of randomness, and the accuracy and precision of the resulting output must be weighed against the assumptions made in developing the system model.

The stages of the simulation development process outlined above are rarely performed in a structured sequence beginning with problem definition and ending with analysis and documentation. A simulation study will likely involve false starts, erroneous assumptions, reformulation of problem objectives, and repeated evaluation and redesign of the model and program before it can properly be used to assess alternatives and enhance the decision-making process.

3. CONCEPTS IN SIMULATION PROGRAMMING

Simulation, as defined in Section 2, is modeling the dynamic behavior of a system as it changes over time. To represent and reproduce system behavior, features not normally found (or adequately emphasized) in most conventional programming languages are needed. These features include:

1. Data representations that support straightforward and efficient modeling of the static structure of systems of interest.
2. Data and control abstractions that permit the facile portrayal and reproduction of system dynamics.
3. Mechanisms oriented toward the study of stochastic phenomena.
4. Mechanisms for collecting, analyzing, and displaying data generated by a simulation program.

There is no one form a simulation language must take to support these features, nor any one accepted method for instantiating them. However, these features are the principal characteristics of most modern simulation programming languages. How these features are elaborated and implemented is what makes particular simulation languages difficult or easy to use, programmer- or analyst-oriented, etc. In this section, we review programming concepts upon which these features are based.

3.1 DESCRIBING STATIC STRUCTURE

The static structure of a simulation model is a time-independent framework within which system states are defined. Dynamic system processes act and interact within the static structure, changing data values and thereby changing system states.

To model static structures, a simulation programming language must be able to:

1. Define the classes of entities within a system;
2. Adjust the number of these entities as conditions vary;
3. Define the attributes that both describe and differentiate entities of the same class;
4. Relate entities to one another and to the environment in changeable yet prescribed ways.

These requirements are not unique to simulation programming languages. They appear in most conventional programming languages and systems that deal with information retrieval and management. Because of this, we will not go into detail on approaches to describing static structure.

The features for defining static structure found in most simulation programming languages are lineal descendants (or at least near relatives) of similar features from a small number of seminal languages. Of recent interest, however, is the fact that researchers have begun applying techniques from object-oriented programming to describing static structure. Further discussion of this topic is given later in Section 4.2.

3.2 DESCRIBING SYSTEM DYNAMICS

The heart of every simulation program, and, consequently, every simulation programming language, is some notion of a *time-control routine*. This component is alternately referred to as a clockworks, a simulation executive, a timing mechanism, a sequencing set, and so forth. Its functions can be described generally as (1) to advance simulation time, and (2) to select a subprogram for execution that performs a specified simulation activity.

In regard to time, system models can be classified as either discrete change, continuous change, or a combination of the two. As a result, the time-control routine of a simulation programming language must be able to support one or all of these views.

3.2.1 Approaches to Discrete-Event Simulation

Generally, three approaches (or *world views*) can be employed in discrete-event simulations: the *process-interaction* approach, the *event-scheduling* approach, and the *activity-scanning* approach.

1. *Process-Interaction Approach.* The activities of a system are modeled as a sequence of processes. This approach concentrates on the progress of entities as they pass through the model, and the model describes the processes that operate on each entity as it moves from creation to termination. Normally, the process-interaction approach requires that a record be maintained for each entity indicating its status, including the conditions that must be satisfied to move to the next process in the model. These conditions may merely be the passage of time, or they may be the results of a set of events involving other entities. This approach is applicable to systems of entities competing for services and is typically used for queuing models.

2. *Event-Scheduling Approach.* The events that model the start and end of an activity are scheduled according to when they should be executed; this is their *event time.* Typically, a list of scheduled events is maintained and ordered by event time. After an event is executed, simulation time is advanced to the event time of the next scheduled event, which is then executed. When more than one event is scheduled at the same event time, the events are ordered according to some metric and executed in turn. This approach is suitable for simulating deterministic systems where the start and duration of activities can be computed dynamically.

3. *Activity-Scanning Approach.* Time is advanced in discrete intervals. Whenever time is advanced, the status of each activity in the model is scanned to determine which can be activated or deactivated. Activities are modeled by events that delineate the conditions under which each activity starts and ends. Thus, scanning the status of an activity means examining the conditions of the events that mark its start and end. If the conditions of an event are satisfied, it is scheduled for execution in that time period. The execution of an event causes state changes that may allow other events to be executed, either during the same time period or at a later one. This approach is well suited for situations where the initiation or duration of activities is determined by changes in system state and cannot be scheduled *a priori.*

3.2.2 Approaches to Continuous Simulation

In a continuous simulation, the state of the system is modeled by dependent variables that change continuously with time; continuous-change variables are also referred to as *state variables.* A continuous simulation is constructed by defining equations for a set of state variables. A set of equations with common variables is known as a system of *simultaneous equations.* The state of the system at a particular time is determined by evaluating the equations modeling it.

An important piece of information in a continuous simulation is the *rate of change* of a state variable. It is often easier to devise an equation for the rate of change of a state variable than an equation for computing the value of the state variable directly. Typically, the rate of change is expressed as a derivative, and the equation for the state variable is expressed as a differential equation, i.e., as a function of its derivative.

Methods for solving simultaneous differential equations have been studied by mathematicians for years. The advent of digital computers and the applications of these systems of equations to continuous

simulations have encouraged considerable research into efficient computations of simultaneous differential equations. The principal difficulty is that a digital computer is discrete in its operations. Thus, solutions obtained using numerical integration techniques on digital computers are only approximations. A description of various numerical integration algorithms can be found in introductory texts on numerical analysis, such as Conte and de Boor (1980). The essence of these techniques is to divide independent variables, such as time, into small slices or *steps*, and then repeatedly solve the differential equations for successive steps. The accuracy of these methods depends upon the order of the approximation method and the size of the step, with greater accuracy resulting from higher-order approximations and smaller step sizes. Since higher-order approximations and smaller step sizes require more computations, a trade-off exists between accuracy and execution speed.

There is another specialized approach to modeling continuous systems, called *System Dynamics* (Forrester, 1961), which focuses on the relationship between the structure of a system and its performance. System Dynamics is oriented toward determining significant trends or behavioral characteristics of systems, rather than precise numerical results. System Dynamics models the rate at which entities flow through a system, as well as dependencies that affect the rate, such as the rate of flow through other systems or feedback from the systems output. For example, System Dynamics could be used to measure the flow of resources through a supply/maintenance system of an air base, where the rate of resources flowing from the supply system is dependent upon the rate of requests by the maintenance system and the rate of resupply.

Computer languages for continuous simulation normally employ either a *block* or *equation* orientation. Block-oriented languages functionally emulate the circuit components of an analog computer, using analog block diagrams for implicitly expressing differential equations. Equation-oriented languages allow equations to be coded explicitly.

There is a set of standards for continuous-system simulation languages (CSSL), developed by a committee of the Society for Computer Simulation (SCi Software Committee, 1967).

3.2.3 Approaches to Combined Discrete-Continuous Simulation

As the name implies, in combined discrete-continuous simulation, dependent variables may change at both discrete and continuous intervals. Typically, the behavior of a system model is simulated by com-

puting the values of state variables at small time steps while computing the values of discrete change variables at event times.

Three fundamental interactions can occur between discrete and continuous change variables.

1. A discrete change in value may be made to a continuous variable (e.g., the sortie rate of an air base can be instantaneously reduced by an enemy attack).
2. An event involving a continuous variable reaching a particular threshold value may trigger another event (e.g., an aircraft cannot stay airborne when its fuel level reaches zero, causing the aircraft to crash).
3. The functional description of continuous variables may be changed at discrete time instances (e.g., the equations for computing the sortie rate of an air base can change as weather conditions change).

A combined simulation language must contain provisions for modeling the interactions described above. Largely, this requires combining the activity scanning and event scheduling world views. Because activity scanning requires examining the conditions of all events at each time change, developing efficient scanning methods is an active area of simulation research (Adam and Dogramaci, 1979).

3.3 MODELING STOCHASTIC BEHAVIOR

An important use of simulation is that of drawing statistical inferences about the behavior of stochastic systems. As a result, many simulation programming languages support mechanisms for modeling statistical phenomena.

In general, statistical phenomena can be perceived in terms of either *uncertainty* or *variability*. Uncertainty enters into a model through statements such as,

> *In situation X,*
> *15 percent of the time Y will occur and*
> *85 percent of the time Z will occur.*

Given that the system is in state X, then a probabilistic support mechanism is required to select Y or Z as the next state. Variability enters into models from statements such as,

> *The time to travel from A to B has an exponential distribution with a mean of 3 hours.*

Thus, the support mechanism must also be able to generate samples from a statistical distribution. Simulation programming languages normally reproduce variability and uncertainty with random numbers. Additional features transform random numbers into variates from statistical distributions or perform sampling tasks.

3.3.1 Pseudo-Random Numbers

Random numbers are needed to introduce uncertainty and variability into a model, but, because of the repetitive nature of experiments that are performed in simulation studies, truly random sequences of numbers are not adequate. One must have reproducible sequences of numbers that are, for all intents and purposes, random so far as their statistical properties are concerned.

Pseudo-random numbers are reproducible streams of numbers generated in such a way as to appear to be random. Since they are not random, but come from deterministic series, they can only approximate the independence of truly random number sequences. For simplicity, we will assume that pseudo-random numbers are statistically independent and uniformly distributed between 0 and 1.

3.3.2 Statistical Samplings

Pseudo-random numbers can be used directly for statistical sampling tasks. They can represent probabilities in a decision sense or a sampling sense. For example, the statement:

> *make decision D_1 60 percent of the time, and make decision D_2 40 percent of the time*

can be implemented by generating a pseudo-random number and testing whether it lies between 0.0 and 0.6. If so, decision D_1 is made; if not, D_2 is. For a sufficiently large number of samplings, D_1 will be selected 60 percent of the time, while the individual selection of D_1 and D_2 will be independent of previous selections.

This approach can be generalized into a table look-up approach based upon a table of cumulative probabilities (Kiviat, 1969). To illustrate, assume we have the table below. A sampling value is selected by generating a pseudo-random number, matching it with a cumulative probability in the first column, and generating a sample value based upon the corresponding entry in the second column.

Cumulative Probabilities	Sample Set
0.20	2.5
0.45	11.8
0.60	20.9
0.77	30.0
0.90	33.3
0.99	50.0
1.00	66.7

In this way, a discrete probability function can be implemented by the following algorithm:

(1) Generate a random number.
(2) Compare this number to successive cumulative probability values until a value is found that equals or exceeds it.
(3) Generate the value from the sample set associated with this cumulative probability.

If the random real 0.20 is drawn in (1), then the value 2.5 is generated; 0.65 generates 30.0; and 0.95 generates 50.0. A continuous probability function is implemented in a similar manner, with the exception of the third step, which becomes:

(3) Generate the result of interpolating the corresponding value from the sample set with the value preceding it. That is, let R be the random real, i be the index of the stopping probability, C_i be the *ith* probability, and V_i be the corresponding sample value. The interpolation formula is then computed as

$$P = V_i + [(R - C_{i-1})/(C_i - C_{i-1})] * (V_i - V_{i-1})$$

Using this method, 0.20 generates 2.5; 0.65 generates 23.6; and 0.95 generates 42.6.

While this type of sampling is useful for empirical frequency distributions, it is less useful for sampling from statistical distributions, such as exponential or normal. To use a table look-up process such as the one outlined above (and sample accurately in the tails of a statistical distribution), large tables would have to be stored and accessed, which would be inefficient in both time and memory, so algorithms rather than table look-up procedures are normally used. Fortunately, the state of the art for sampling from statistical distribution functions is quite advanced. Textbooks on simulation usually include one or more chapters devoted to random and statistical sampling procedures and

algorithms (Fishman, 1978; Law and Kelton, 1982; Bratley, Fox, and Schrage, 1983). A survey of random variate generation techniques by Schmieser (1980) contains over 300 references.

3.3.3 Controlling Randomness

In conducting a simulation study, there is interest in control and precision as well as accuracy or representation. The topics dealt with so far have largely been concerned with representation.

Control is necessary when one is using a simulation to test and compare alternatives, such as variate rules and procedures, quantities of equipment, or availability of resources. When several simulation runs are made that differ only in one carefully altered aspect, it is important that all other aspects remain constant, i.e., one must be able to introduce changes only where they are desired. This is one of the reasons for requiring reproducible streams of random numbers. A feature that aids in this respect is the provision for multiple random-number streams. Having more than one stream enables parts of a model to operate independently, as far as data generation is concerned, and not influence other parts.

Another situation in which one needs to control the generation of random numbers is when doing so will reduce the variability of simulation-generated performance figures. Variance reduction techniques are discussed in Fishman and Kiviat (1967) and further in Bratley, Fox, and Schrage (1983). The reduction of sample variance is actually a statistical rather than a programming problem with the exception that the programmer should be able to control the generation of pseudo-random numbers if required. For example, a simple technique to reduce variance is to use antithetic variates in separate simulation runs, where $1 - R$ is the antithetic variate of the pseudo-random number R. Simulation programming languages that support statistical modeling should likewise support at least this level of control.

3.4 DATA COLLECTION, ANALYSIS, AND DISPLAY

The principal output of simulation experiments is statistical measurements. Such quantities as the average length of jobs through a maintenance system or the percentage operational-effectiveness of a weapon system are typical. Since the objective of a simulation is to study the performance of a system as it changes over time, the collection of data from a simulation model, its analysis, and its display are all key elements of a simulation study. System dynamics can be traced

by plotting relevant object attributes and relations as they change over time, or by displaying representative icons on a high-speed graphics device. Aggregate performance can be studied by looking at statistical analyses of simulation-generated data, such as means, variance, and minima and maxima.

The best that one can do about the mechanisms controlling data collection, analysis, and display is to make them unobtrusive. While these mechanisms are necessary, the portions of a simulation program that comprise these mechanisms are not part of its modeling logic and should not obscure the operations of the model in any way. Ideally, a simulation language should automatically produce all data collection, analysis, and display. Unfortunately, this is not an ideal world; display formats differ among organizations, display media vary, etc. In addition, efficiency is gained by analyzing only the data that are of direct interest to the simulation study.

3.4.1 Data Collection Facilities

Simulation programming languages that support data collection facilities normally do so in a declarative manner. The analyst adds statements to the simulation program specifying what variables are of interest and what type of data should be collected.

Simulation is a statistical tool, and so statistically useful data are required in its effective use. To compute all the statistics an analyst might want about a simulation variable, some simulation languages support a variety of approaches to collecting data on a variable. This can include counts of the number of times a variable changes value, sums, sums of squares, maxima and minima of these values, histograms over specific intervals, cross-products of variables, time-integrated sums and sums of squares for time-dependent data, and time series display.

3.4.2 Statistical Output Analysis

In simulation studies, inference or predictions concerning the behavior of the system under study are based on experimental results obtained from the simulation. When the simulation model contains stochastic variables, the outputs from the simulation are treated as observed samples of stochastic variables. Because the simulation model is built to provide results that resemble the real system, statistical analysis of the outputs from a simulation is similar to the analysis of empirical data obtained from the system. The main difference is that the simulation analyst has more control over the running of the

simulation and can design experiments to obtain the specific output data necessary to answer pertinent questions relating to the system under study.

Two types of questions are related to the output of a simulation model:

1. What is the inherent variability associated with the simulation model?
2. What can be inferred about the real system from the simulation model?

The first relates to understanding the sensitivity of the model and verifying that it performs as intended. The second relates to the validity of the model and its variance from the system.

Answering the first question involves a detailed statistical output analysis to obtain information on the precision and sensitivity of the model. Typically, this requires running the simulation repeatedly or for longer periods of time. Techniques for reducing biases due to initial and final conditions, for estimating the proper output samples to collect and simulation runs to perform, and for reducing the variance of stochastic variables have been explored extensively in simulation research (Bratley, Fox, and Schrage, 1983; Law and Kelton, 1982; Welch, 1983).

Answering the second question is problematic largely because it is system- and model-specific. A coarse level of validation is simply to verify that the simulation runs in a "reasonable manner." A finer level of validation can be done by establishing *certainty intervals* for specific outputs through statistical methods such as the *t*-test (Blum and Rosenblatt, 1972; Chao, 1969). The level of validation for a given simulation depends upon the amount of information available on the system under study.

3.4.3 Display and Graphics

Simulation programs contain and generate information—a lot of information. Most simulation programming languages support mechanisms that allow users to collate and filter data of interest collected during a simulation run, then display that data in a meaningful format. Early insights into this level of display can be found in Kiviat (1966).

An emerging display application is that of interactive graphics interfaces. In general, there are two uses for such interfaces:

1. By associating graphical representations to data of interest, analysts can visually monitor the movement of the system model through simulation time, alternately verifying the accuracy of the model and gleaning insights into the system's behavior;
2. By associating graphical representations to modeling concepts, analysts can develop, experiment with, and modify system models through the use of graphic interface tools, reducing or even eliminating the coding effort required to implement the model into a simulation program.

The potential of higher-order display facilities to support the communication of information is recognized in most simulation languages and projects. Unfortunately, the full manifestation of this support has yet to be achieved. Some commercially available expert system shells, such as ART (Inference, 1986) and KEE (IntelliCorp, 1985a), provide noteworthy interactive graphics facilities. Graphic aids for simulation are also an area of current RAND research (Rothenberg, 1986).

3.5 OTHER CONSIDERATIONS

There are several other issues of concern in simulation programming that should be addressed in a simulation programming language, namely, start-up and look-ahead, debugging and explanation, and usability. Below, we provide a brief discussion on these topics. Additionally, note that some are research topics and not normally found as features of contemporary simulation programming languages.

3.5.1 Start-up and Look-Ahead

The initial conditions for a simulation program may cause data collected from that program to be different from that collected after a start-up period. Earlier we pointed out that the state of a system has both a static and dynamic structure. The static component consists of the system's entities, their attributes, and relations involving entities; the activities in which these entities are engaged make up the dynamic component. When we run a simulation program, however, the start state (unless otherwise provided for) has only a static component. While activities will begin in this state, no activities will be "in progress." Thus, before we can monitor changes in the system model, we must run the simulation program for some indeterminate length of time until it reaches *equilibrium* (Conway, 1962), i.e., a *steady state* that accurately replicates the dynamic structure of the actual system.

For example, consider a maintenance system at an Air Force base. Such a system will normally have several jobs pending and others in progress at any given point in time. If a simulation program for this system were started in an "empty and idle" state (i.e., with no jobs pending or in progress), the time for the first few jobs to pass through the maintenance system would not accurately reflect normal delays (i.e., by jobs already in the system). If the purpose of the simulation study is to measure the average time a job takes to pass through the maintenance system, then the inaccuracy in the time of these initial jobs would invalidate the results of the study. Even if not directly being monitored, they could still propagate errors to other aspects of the model.

Naturally, the ideal would be to start the simulation program in a steady state, sampling from the steady-state distributions that underlie the simulation model and setting the start state accordingly. Unfortunately, knowledge of the steady-state distribution would preclude the need for the simulation. There are essentially two approaches taken in determining the start-up policies for a simulation program:

1. Start "empty and idle," anticipating erroneous data;
2. Start "close" to the steady state, reducing the margin of error.

Anticipating the effects of start-up error becomes increasingly difficult for larger models, so the second approach, though not without its own disadvantages, is normally preferred. A summary of past research into possible start-up policies can be found in Wilson and Pritsker (1978).

Another aspect of start-up is the ability to save the state of the system model during a simulation run and then reinitialize to that state at a later time. This could be used in developing and reusing a steady-state approximation for a given simulation. It could also be used to support a *look-ahead* ability for *intelligent exploration* (Rothenberg, 1986). Such an ability would allow analysts running interactively to readily explore alternative scenarios without repeatedly having to start up the simulation program "fresh." The major consideration in supporting such a capability is balancing the memory cost of saving state against the computational cost of restoring state.

3.5.2 Debugging and Explanation

Debugging a simulation program is made difficult by two factors: (1) flow of control can be stochastically determined; and (2) flow of control of events passes through the time-control routine. The first factor increases the complexity of program validation; the second factor obscures the record of control flow for system activities. While

approaches to addressing the first factor are beyond the scope of this discussion, an approach to the second factor is to trace system activities and maintain a record of state changes. This record could then be accessed and investigated when an error occurs during execution.

These same facilities are needed for supporting an explanation capability. Explanation is a capability that supports the model user, rather than the model builder, and in some ways is tied into data collection and analysis. It is intended to aid the analyst in understanding system behavior. For example, the analyst could ask how the simulation program went from one state to another. The explanation mechanism might respond with the sequence of events that led up to the state change and the conditions that initiated these events. If the simulation program is linked with a look-ahead facility, the analyst could then return to a previous state and explore a new scenario interactively.

3.5.3 Usability

Beyond the aspects of simulation programming discussed already, there are also a number of nonoperational aspects in regard to simulation programming and languages. An analyst is interested in program readability. Communication of the structure, assumptions, and operations of the model implemented in a simulation program is an aid to using both the program and model correctly. An analyst is also interested in execution-time efficiency. Simulations can require large numbers of experimental runs, and the cost per run must be low enough to make a study economical. An analyst must also balance the cost of executing a program against the cost of producing, updating, and maintaining it. High-level modeling languages may compile and execute less efficiently than simpler languages, but they narrow the gap between system model and simulation program, making otherwise difficult problems tractable and improving readability. If total problem-solving time is important, as opposed to computer time costs, the evaluation criteria change.

4. CONCEPTS FROM AI AND OOP

The emphasis in current simulation research focuses on the application of techniques from artificial intelligence (AI) and object-oriented programming (OOP) (Oren, 1977; Klahr and Faught, 1980; Reddy and Fox, 1982; Klahr, McArthur, and Narain, 1982; Klahr et al., 1984; Steeb et al., 1984; Shaw and Gains, 1986; Rothenberg, 1986). In this section, we provide an overview of basic concepts in these areas and examine their strengths and weaknesses in regard to simulation.

4.1 RULE-BASED PROCESSING

Artificial intelligence is such a broad field that a general discussion of topics of interest to simulation programming is beyond the scope of this report; interested readers are referred to Nilsson (1980) and Rich (1983). As much of the current interest is directed toward expert systems technology, we will focus our discussion on aspects of this subfield of AI. In particular, we will focus on concepts in *rule-based processing* and *production system architectures.*

Although production systems have their roots in mathematics (Post, 1943), the use of rule-based processing and production systems in AI originated with the seminal work of Newell (1967), who proposed them as a formalism for information processing theories of human problem-solving behavior. Since then, production systems have received considerable attention in the areas of expert systems (Hayes-Roth,Waterman, and Lenat, 1983; Buchanan and Shortliffe, 1984) and cognitive psychology (Klahr, Langley, and Neches, 1987) and have shown applicability to such tasks as general problem-solving, knowledge-based programming, planning, natural language understanding, cognitive modeling, and learning. They are widely accepted in AI as information-processing models well-suited for decision-making tasks in complex and poorly defined domains.

4.1.1 Basic Concepts

In general, a production system consists of three parts: (1) a set of *facts*, which represents system state; (2) a set of *rules*, which maps states to states; (3) an *inference engine*, which controls the applications of rules with the objective of moving the system from its initial state to some goal state. A rule consists of two basic components, an

antecedent and a *consequent*, each of which describes a partial configuration of system state (as represented by facts). Together, the antecedent and consequent of a rule describe a set of state transitions, i.e., *from* the states corresponding to the antecedent *to* the states corresponding to the consequent.

The inference engine applies rules in a *forward* or *backward* manner, essentially moving the system from the states described by the antecedents to those described by the consequents or vice versa. (This is alternately referred to as *forward-chaining* and *backward-chaining*.) In either direction, the inference engine can be described as *searching* for a path between the initial state and the goal. In the forward direction, this search proceeds from the initial state to the goal; in the backward direction, it proceeds from the goal to the initial state. Forward-chaining systems are also said to be *state-driven*, while backward-chaining systems are said to be *goal-driven*.

Forward-Chaining Systems. In a forward-chaining production system—as exemplified by the OPS family of production system architectures (Brownston et al., 1985)—rules are often called *if-then* or *condition-action* rules, where the antecedent consists of a set of conditions on system state and the consequent consists of a set of state-changing actions. The inference engine processes rules according to a *recognize-act* cycle, which consists of three distinct phases:

1. The *match* phase, in which all rules whose conditions are satisfied by the current system state are collected into a *conflict set*. (Actually, the conflict set consists of rule *instantiations*, recording the conditions of the match. If the same rule can be satisfied in several ways, each *instantiation* is added to the conflict set.)
2. The *conflict resolution* phase, in which one or more instantiations are selected from the conflict set.
3. The *act* phase, in which the actions of the selected instantiations are applied, changing system state.

At the end of a cycle, changes to system state allow other rules to be satisfied and applied on the next cycles, and so forth, until a terminating condition (the goal state) has been reached. Note that this account ignores many of the details as well as the many variations that are possible within this framework (e.g., different matching, resolution, and application strategies). More detailed discussions can be found in Waterman and Hayes-Roth (1978).

Backward-Chaining Systems. Backward-chaining production systems—as exemplified by PROLOG (Colmerauer, Kanoui, and Van

Caneghem, 1983)—were derived largely from work done on automated theorem-proving. As a result, use of such systems is typically referred to as *logic programming* and rules as *implications*. A rule has two parts, a *head*—a unary clause representing an implicit set of facts—which corresponds to the rule's consequent, and a *body*—either a unary clause or a conjunction of clauses—which corresponds to the rule's antecedent.

The inference engine processes rules using *resolution* and *unification* for the purpose of satisfying a goal. When the goal is a unary clause, the inference engine operates by finding all facts and rules that could be used to satisfy the goal. For a fact to satisfy, it must unify with the goal; for a rule to satisfy, its head must unify with the goal and its body must be successfully resolved. When the goal is a conjunction of unit clauses, the inference engine operates by resolving each clause in turn, treating each as a subgoal. In this way, the inference engine can be thought of as operating in a top-down, recursive manner.

Unification is a pattern-matching technique for use in situations where both the "pattern" and the "datum" may contain variables. Unification is necessary to a backward-chaining system because the inference engine is sometimes required to match goals, which are patterns of constants and variables, against the heads of rules, which are also patterns of constants and variables. Unification takes two patterns and determines whether it is possible to bind values to the variables in such a way as to make both patterns ·ual. As some patterns can unify in more than one way, the unifier for a backward-chaining system will typically return a set of all possible variable bindings that allow two patterns to unify.

If unable to resolve its current goal, the inference engine will backtrack to the last point at which it could have used some alternative resolution method and continues, using the alternative. Such backup points are generated when more than one fact or rule could possibly be used to satisfy a unary goal. As with forward-chaining systems, this is a gross simplification and there are many variations on this theme; see Kowalski (1979) and Campbell (1984).

4.1.2 Strengths and Weaknesses

There have been a number of claims about the advantages of rule-based processing. Unfortunately, many of these claims have a rhetorical quality to them, so we will not repeat them here. Instead, we will focus on aspects of rule-based processing that have advantages (or disadvantages) to simulation programming.

One strength of production systems is that they characterize a higher-order computing architecture than conventional von Neumann machines. Much of the behavior of a rule-based system is implicit in the organization of rules and the operation of the inference engine. This means that such high-level behavior does not have to be explicitly coded by a programmer or, later, decoded by a reader. This does not mean that nonprogrammers can automatically read rule-based programs, just that such programs are formalized using a higher-level of abstraction; the reader must still understand the operation of the particular production system architecture. In addition, automated tools for validating a system model, or tools to help encode the model or explain its performance, can also operate at a higher-level of abstraction and thus offer more sophisticated responses.

The dynamic structure of a system model is described by the activities in which the entities of the system are engaged. Sometimes the behavior of entities can follow a predetermined sequence. However, in stochastic systems, it is not always possible to predict such exact behavior, so entities must be able to react to the situations in which they find themselves. This is in part the motivation behind the activity-scanning approach to discrete-event simulations (see Section 3.2.1). A strength of forward-chaining production systems (or state-based processing, in general) is the close correspondence between the control and application of rules by the inference engine and the activation of events by the timing-control routine.

Despite its naturalness, few simulation languages support activity scanning for performance reasons. The naive approach to implementing activity scanning is with a control loop similar to that described by the recognize-act cycle, testing the conditions of every activity at the start of each cycle. There are several drawbacks to this approach:

1. Execution time performance degrades as the number of activities in the simulation increases.
2. Activities with similar conditions will cause redundant computations (e.g., if one activity depends on conditions X and Y and another on X and Z, X will be computed twice).
3. If system state changes in small steps on each cycle, many of the "scanning" computations performed at the start of one cycle will have the same results on the next cycle, introducing an additional source of redundant computations.

Similar problems were recognized in early production system architectures but were overcome by the development of the RETE Matching Algorithm (Forgy, 1982). Essentially, this algorithm works by

1. Merging common conditions, such that the effort to test for any one condition (even if duplicated by several rules) is only done once;
2. Off-loading the cost of the matching process to the state-changing operations (scanning for any one condition is done only when a corresponding state change occurs).

Using the RETE algorithm, execution time is not proportional to the number of rules in the system. The increase in efficiency has been sufficient to spawn commercially viable rule-based languages, such as ART (Inference, 1986) and Knowledge-Craft (Carnegie Group, 1986). A technique similar to the RETE Matching Algorithm could also be used to efficiently support the activity-scanning approach in a simulation programming language.

In regard to the strength of backward-chaining systems, consider the state of a system model as a data base and a backward-chaining production system as a front end to this data base. The rules of a backward-chaining system can be viewed as defining an abstract layer of facts on top of the concrete layer of facts represented by system state. The head of a rule implicitly represents a set of facts, which are instantiated (through unification) when the body of the rule is resolved. Resolution of the body of a rule may require the use of other abstract facts but eventually terminates at the level of concrete facts. An aspect of unification is that components of the concrete facts can be passed back up the initial query via a sequence of variable bindings. In this way, backward-chaining supports a notion of *deductive information retrieval.* The advantage of this is that the abstract facts do not need to be explicitly defined but can be inferred from system state.

There are several noteworthy weaknesses to production system architectures (but we will generalize to just two). First, both forward- and backward-chaining production systems, almost by definition, are interpreted, where the inference engine is the interpreter. This is a problem because interpretation inherently detracts from efficient execution. Despite the fact that "compilers" exist for some production system architectures, there is still sufficient overhead required to noticeably degrade performance in large systems. Second, the control structure of production system architectures is not particularly flexible, making certain computations difficult to describe. As a result, programmers are often forced to adopt a stilted and inefficient coding style, which detracts from the readability and utility of their programs.

4.2 OBJECT-ORIENTED SIMULATION

Object-oriented simulation refers to the application of object-oriented programming techniques to the encoding of computer simulations. Although the original motivation for the OOP methodology came from SIMULA (Dahl and Nygaard, 1966; Dahl, Myhrhaug, and Nygaard, 1970), a language intended for use in developing simulations, object-oriented simulation (at least in the United States) did not begin to build a following until the 1980s when OOP was popularized by the Smalltalk-80 language (Goldberg and Robson, 1983).

The basic tenet of OOP as a programming methodology holds that the behavior of a software system is distributed among the conceptual entities represented by the system and described as the behavior of such entities in response to particular situations (Stefik and Bobrow, 1986). Thus, an *object* is an active entity that combines the properties of procedures and data, i.e., it can save local state and perform computations. This characterization contrasts OOP with traditional, procedure-oriented programming, in which the behavior of a software system is distributed among data-independent procedures and described in terms of these procedures as applied to a passive data set. The appeal of OOP is to applications in simulation where entities of a system instigate actions, e.g., command units in military systems.

A marked characteristic of OOP is the wide diversity in its definition. It has been said that there are as many different views of what OOP is as there are computer scientists (Pascoe, 1986). The description given below attempts to generalize away from any particular instantiation.

4.2.1 Basic Concepts

An object is a dynamic processing entity with both data and procedural attributes. The data attributes of an object define its local state, while its procedural attributes define its response to particular situations. An object is generated as an instance of one or more *classes*. A class is a generic description of a set of similar objects, defining the data and procedural attributes to be associated with those objects when generated.

Classes are arranged in a directed lattice called a *class hierarchy*. A class can be linked to one or more classes above it in the hierarchy and is said to inherit from those classes. An object system that allows a class to inherit (directly) from just one other class is said to have *single inheritance*; inheritance from more than one class is called *multiple inheritance*. The data and procedural attributes of a class consist of

those attributes explicitly declared for it, plus some combination of the attributes inherited from the classes above it. Attributes declared for a class normally have precedence over similar inherited attributes, allowing a class to be viewed as a specialization of the class from which it inherits. Objects that are instances of a particular class are also considered to be instances of the classes from which that class inherits.

Object behavior is described in terms of the procedural attributes associated with an object. Each such attribute is a function that implements the response of an object to a particular situation. In some systems, objects are allowed to respond asynchronously, thus readily supporting notions of parallel processing. The "situations" to which an object can respond can be represented as the invocation of a procedure, or, as in the case of many contemporary OOP systems, the receipt of a *message*, where a message denotes an imperative to an object directing it to take an action on some set of arguments. In a less restricted definition, a "situation" could correspond to a particular configuration of an object's local state, the global state of the program, the local state of other objects, or some combination of all of these.

Processing in an object system is controlled by stimulating situations to which objects must respond. In some object systems, this is done by invoking a procedure or sending an object a message. Taking the wider definition of what constitutes a situation as suggested above, this can also be done by making changes to local or global state. Process behavior is dependent upon the particular objects involved in the process; the same situation can invoke different behavior given different objects.

4.2.2 Strengths and Weaknesses

Object-oriented programming is not so much a programming technique as a programming methodology, i.e., a way of approaching the process of software design and implementation. Its primary strength is that it encourages sound, mature software practices, such as encapsulation, data abstraction, and software reusability. Because of the complexity of computer simulations, they can only benefit from the influence of such practices.

Encapsulation is a method for minimizing interdependencies among separately written program modules by stipulating that each module provide an external interface by which it can be accessed. This hides the implementation of the module from its application and allows changes to be made to a module without changing the modules that use it. *Data abstraction* is essentially the application of encapsulation. It means that a programmer can use and combine the data types and

operations of several modules without needing to know how the modules are implemented, with the result that changes to a module do not necessitate changes to programs that use it. *Software reusability* is likewise tied to data abstraction and encapsulation. It simply means that software modules, because of their relative independence, can be reused in a number of applications.

Another strength of OOP is the fact that it reduces the distance between developing a system model and encoding that model into an executable computer simulation. Modeling concepts can be mapped readily, and almost directly, into the programming concepts of an OOP language. Entities and their attributes correspond to objects, while activities correspond to the procedural attributes of objects.

Unfortunately, the strengths of OOP as a programming methodology also contribute to a significant weakness, i.e., slow execution-time performance. The simplest way for a programming language to support the notions of encapsulation, data abstraction, and reusability is to postpone until execution time many decisions that would normally be made at compile time (e.g., determining the location of an operator's definition and the storage requirements of data). This postponement, which is known as *late binding*, has the predictable effect of reducing execution speed.

Perhaps a worse weakness is that there is still no clear agreement on just what OOP means. We have presented here a brief, broad, and liberal view of OOP, but there are several extreme views with conflicting opinions, which leads to much confusion among those without a background to appreciate the tentativeness of these views. For instance, the majority of existing OOP systems claim that message passing is the *only* means of exciting object behavior. The truth is that message passing is simply a useful process metaphor, but other useful metaphors also exist, such as that of *generic operations* as exemplified in CommonLoops (Bobrow et al., 1986). Additionally, many claims are being made about the power of OOP to solve the intractable problems of software engineering. Such claims only lead to false expectations and disappointment; they can be likened to similar claims made for expert systems.

Simply put, OOP is a structured way of thinking about the software development process, a way that encourages good practices. It does not, unfortunately, ensure that these processes will be utilized to the best effect. Object-oriented programming certainly has a place in simulation programming, but it earns its place because it is a mature software methodology that encourages sound attitudes toward program development, not because objects and message passing are possessed of magical powers to cure all programming ills.

5. SIMULATION PROGRAMMING LANGUAGES

The widespread use of simulation as an analysis tool has led to the development of a number of languages specifically designed for simulation. Simulation programming languages assist the development of simulation models through their "world view," they expedite computer programming through their specialized high-level programming constructs, and they encourage proper model analysis through their data collection, analysis, and reporting features. Particularly important aspects of simulation programming languages (outlined earlier in Section 3) include modeling a system's static and dynamic structure, statistical sampling, and data collection, analysis, and display; additional considerations include start-up policies, debugging and explanation, and usability.

Below is a brief description of nine current and proposed simulation languages: Simscript, SIMULA, GASP IV and SLAM, ROSS, RAND-ABEL*, ERIC, SimKit, ModSim, and ROBS. We have intentionally left out several other well-known simulation languages, such as GPSS and SIMLAB, because we feel that they are either conceptually close to other languages being presented or ill-suited for application to military simulation. Languages are presented roughly in chronological order.

5.1 SIMSCRIPT

Simscript is a general-purpose programming language with built-in support for discrete-event simulation as well as some support for continuous-valued variables. Simscript was originally developed at RAND in the early 1960s (Markowitz, Hausner, and Karr, 1962) as a simulation-oriented preprocessor to Fortran. Simscript II.5 (CACI Inc., 1976a) is the latest outgrowth of the Simscript family. Simscript is a noteworthy language to review. Despite its age and its eccentricities, it is still actively used and supported.

The focus of Simscript is the characterization of the status of a system in terms of *entities*, *attributes*, and *sets*. Essentially, an entity is an abstract data structure, and its attributes are the fields within this structure; sets are doubly linked lists. A Simscript program consists of four major components: a *preamble*, a *main* routine, a set of *event* routines, and a set of supporting subprograms. The preamble defines the structure of the model in terms of its entities and their attributes, the sets to which entities may belong, the events that can occur, and the

statistical information to collect. The main routine initializes the model and initiates the execution by scheduling the first event. The event routines are time-dependent subprograms that indicate the actions to be taken when an event occurs.

Simscript models time using an event-scheduling approach, as described in Section 3.2.1. System dynamics are modeled by event routines that describe actions to be taken when an event occurs. Events are declared in the preamble and scheduled dynamically at execution time. The main routine schedules the first series of events and then executes a START SIMULATION statement, which advances time to the next event and executes its actions. An event routine normally schedules more events and may likewise reschedule itself. A Simscript simulation is driven by an event list that is ordered by time and priority. The system will execute the event routines for all events scheduled for a given time in priority order. When no events scheduled for the current time remain, time is advanced to the time of the next event and the process is repeated. When the event list becomes empty, control is returned to the point where the START SIMULATION statement was called.

There are two types of functions in Simscript, *right-handed* and *left-handed*. Right-handed functions conform to the semantics typically associated with subroutines, i.e., they compute values. Such functions normally appear on the right-hand side of assignment statements, hence the name. Left-handed functions appear on the left-hand side of assignments; i.e., they receive the assigned value as input, and (instead of storing that value) they perform computations on it. This idea is extended to variables and attributes, providing a notion of *monitored* variables and attributes. (These ideas have reemerged in current work in applying OOP to knowledge-based programming as *access-oriented* programming (Stefik and Bobrow, 1986).)

Simscript contains mechanisms that permit the definition of statistical quantities in the preamble and the automatic collection of statistical output during the execution of the simulation. These facilities operate by automatically putting monitors on variables and attributes of interest. These monitors keep track of accesses and changes, recording such things as the mean, maximum, and standard deviation of their value, number changes, and so forth. The time interval at which changes occur can also be monitored and made a factor of the collected statistics, e.g., allowing one to calculate such things as the average size of a queue over the duration of the simulation run. Simulation results are output according to a user-defined format.

While Simscript is essentially event-oriented, recent extensions to the language allow systems to be described in terms of *processes* and

resources (CACI Inc., 1976b). Processes and resources are special forms of entities. A process is a subprogram that can be initiated, suspended, resumed, and terminated. Processes can request and relinquish resources; they are made to wait if a requested resource is not available.

Simscript has had a long history of introducing innovations in simulation technology. However, as a programming language, it has many severe handicaps. Its uncontrolled and inconsistent syntax, complex storage management facilities, inadequate control mechanisms, and eclectic choice of features hinder the development and maintenance of large systems (Bratley, Fox, and Schrage, 1983). Much of this is due to the fact that CACI has a large customer base that would not tolerate changes to the language that would necessitate recoding simulation programs that already work. Hence, when additions are made to the language, old or archaic constructs remain as well, to maintain upward compatibility.

5.2 SIMULA

SIMULA is widely accepted as the first language to instantiate the concepts of object-oriented programming. SIMULA, which stands for SIMUlation LAnguage, was originally intended as a language for discrete-event simulations. (Its approach to simulation was influenced largely by Simscript I (Markowitz, Hausner, and Karr, 1962).) SIMULA I (Dahl and Nygaard, 1966) extended ALGOL 60 to support list processing facilities and time-ordered coroutines, called *processes*. Later, SIMULA 67 (Dahl, Myhrhaug, and Nygaard, 1970) generalized the coroutine concepts of SIMULA I into the concepts of classes, class hierarchies, and instance objects.

Two standard classes defined by SIMULA are used for discrete-event simulation, SIMSET and SIMULATION. The SIMSET class simply defines the concept of linked lists, while SIMULATION defines concepts necessary for process-oriented simulation. Time is represented by a *time axis*, a list of *event notices*. An event notice has two attributes, a reference to a process and a time of occurrence. Simulation time is the time of the next event notice on the time axis. Time is advanced by removing an event notice from the axis and resuming its process. The SIMULATION class supports several procedures for scheduling processes that operate by automatically scheduling events that point to the processes.

SIMULA provides procedures for operating on objects that are used to define standard input and output procedures. (In ALGOL, on which

SIMULA is based, input and output procedures are not defined.) There are additional procedures for generating pseudo-random numbers as well as drawing from empirical distributions. Only two rudimentary procedures, one for calculating time-integrated values and another for accumulating histograms, are provided for gathering statistical data.

Despite the basic elegance of classes, SIMULA is complex and difficult to use. To write even the simplest simulation with the predefined SIMSET and SIMULATION, the programmer must have a working knowledge of ALGOL 60, which has never had a large following in the United States; he must also wade through a sea of preliminary definitions, such as *links, linkages, heads, processes*, and so forth. There is only minimal support for accumulating statistical data and outputting formatted reports. In summary, SIMULA is not as good a simulation language as it might have been. Additional details on simulation in SIMULA can be found in Birtwhistle et al. (1973) and Franta (1977); a more detailed criticism of SIMULA can be found in Bratley, Fox, and Schrage (1983).

5.3 GASP IV AND SLAM

GASP IV (Pritsker, 1974) is an event-oriented simulation package consisting of some 30 Fortran subroutines and functions, each of which performs a required simulation activity. GASP IV is distributed by Pritsker and Associates, Inc. Because GASP IV is an extension to Fortran, it is less elegant and expressive than other languages specifically designed for simulation, but, on the plus side, it is relatively simple to learn, widely available, and executes at the speed of Fortran.

A GASP IV discrete-event simulation model views a system as consisting of *entities*, their associated *attributes*, and *files*, which contain entities with common characteristics. (Files are similar to sets in Simscript and lists in SIMULA.) All files are stored in a single master array, called NSET. An execution routine, called GASP, automatically performs such activities as selecting the next event from the event list and advancing simulation time. The modeler must write a main program, an initialization subroutine INTLC, an event handler subroutine EVNTS, a set of event subroutines, and a report subroutine OTPUT. In the main program, the total amount of storage required for a simulation is specified by dimensioning NSET, the user-defined variables are initialized, and then the statement CALL GASP is executed to begin the simulation. Subroutine INTLC is used to initialize user-defined variables at the start of each simulation run. Subroutine

EVNTS, which is called by GASP with the event type of the next event, uses a computed "goto" to pass control to the appropriate event routi... GASP IV supplies a standard output report at the end of the simulation and uses OTPUT to supply additional output.

SLAM (Pritsker and Pegden, 1979) is a descendent of the GASP family and is also supported and distributed by Pritsker and Associates, Inc. SLAM extends GASP by including support for process-oriented and continuous simulation. Event orientation in SLAM is similar to that in GASP IV. Using a process orientation, a modeler combines a set of standard symbols, called *nodes* and *branches*, into an interconnected *network* structure to represent the system of interest graphically. This representation models the processes through which entities in a system flow. After the network model of the system has been developed, it is translated into an equivalent set of SLAM program statements.

The real appeal of SLAM is the diversity of modeling approaches it supports. An analyst can build discrete-event simulations using an event or process orientation (or both), continuous simulations using differential or difference equations, or combined simulations using all these elements.

5.4 ROSS

ROSS (McArthur, Klahr, and Narain, 1985) is an object-oriented simulation system developed at RAND under Project AIR FORCE. ROSS was intended to represent a new approach to simulation languages, employing concepts from AI to make it easier to build, modify, and understand combat simulations.

Although ROSS is an acronym for Rule-Oriented Simulation System, its concept of "rules" is not related to the description of rule-based processing found in Section 4.1. ROSS is noteworthy because of its use of the object-oriented concepts developed in ACTORS (Hewitt, 1977) and DIRECTOR (Kahn, 1979). ROSS objects, called *actors*, are arranged in an inheritance hierarchy rooted at the predefined object SOMETHING. The data components of a ROSS object are called *attributes*, and the procedural components *behaviors*. A behavior of an object is activated through message passing. Each behavior specifies a pattern, called its *message template*, which designates the general form of messages that can activate it. When an object receives a message matching a behavior's message template, the actions associated with the behavior are executed.

ROSS supports discrete-event deterministic simulation. ROSS models the advance of simulation time with a predefined clock object, which advances in discrete time steps. Objects schedule events, called *plans*, as messages to be sent at a particular time. On each *tick* of the clock, time is advanced to successively scheduled plans, sending the appropriate messages to the objects designated by each plan, until a *ticksize* of seconds has passed. Control is returned from the timing routine after a specified number of ticks.

Recent extensions (Cammarata, Gates, and Rothenberg, 1988) to ROSS support a flavor of continuous change variables. With these extensions, a user declares which attributes of an object are dependent on time, called *history-dependent*, and which attributes are used in computing their values, called *history-affecting*. The value of history-dependent attributes is computed when requested, as a function of *clock* time. If the value of a history-affecting attribute is changed, then the value of all history-dependent attributes are updated automatically to reflect the change. The user can monitor for time-dependent conditions with *demons*. A demon estimates the time the designated condition will occur and schedules an event to be resumed at that time. If the condition exists when the demon is resumed, then it executes a specified action. If not, it repeats this process, reestimating the time of the event. The user must also declare *demon-affecting* attributes, which, when changed, cause all demons to be resumed.

ROSS is implemented as an extension to LISP rather than a distinct programming language. Many aspects of LISP are carried over into the design of ROSS, and, as a result, one may intermix ROSS and LISP code freely in a ROSS simulation. Actually, one is necessarily forced to mix ROSS and LISP. ROSS provides no mechanisms for program control other than message passing; for conditional and iterative control, LISP must be used. The unfortunate result is that ROSS programs tend to take on the appearance of LISP programs, which non-LISP programmers find difficult to read and even more difficult to write. Another problem is that the execution speed of ROSS simulations is bound to the speed of LISP. As a dynamic language, LISP is inherently slower than static languages, such as Fortran, which is commonly used in military simulations.

5.5 RAND-ABEL*

RAND-ABEL (Shapiro et al., 1985) is a general-purpose programming language for modeling command decisions. It was developed by the RAND Strategy Assessment Center (RSAC) for use in building war

games at the national command level as part of the RAND Strategy Assessment System (RSAS).

Before RAND-ABEL, RSAC used ROSIE• (Kipps, Florman, and Sowizral, 1987), an English-like programming language for applications in AI. Because of ROSIE's dependency on LISP, its execution performance was unacceptably slow. RAND-ABEL was developed as a C-based simplification of ROSIE. RAND-ABEL began life as a preprocessor to C but has matured to the point that it can be considered a language that compiles to C. Although the initial design of RAND-ABEL was partially driven by expedience and the need for short-term utility, later efforts at structured redesign have largely corrected original inconsistencies and restrictions and enhanced the language with several unique features. In its present form, RAND-ABEL is a block-structured, procedure-oriented language with such features as a restricted English-like syntax, a form of coroutining facility called *scripts*, hierarchical name spaces, and a powerful data and control abstraction facility called *decision tables*.

By itself, RAND-ABEL cannot be called a simulation language. The RSAC holds the view that command decisions can be modeled separately from the time delays involved. As a result, RAND-ABEL provides no built-in time control facilities. Instead, the concept of a "wake-up rule" is used to coordinate the activation of events with the progress of an external system clock, while scripts are used to model the events of decision-making activities. Scripts are a generalized coroutine facility in which a special "monitor" coroutine controls all sequencing. A script (coroutine) detaches by resuming the monitor process, which examines the wake-up rules to select the next script to resume. When no wake-up rule is applicable, a clock advance occurs. (In the RSAS, this is accompanied by execution of the Force simulation, but from RAND-ABEL's point of view this is incidental.) The wake-up rules are again tested, and so on.

One of the more unique and notable programming constructs supported by RAND-ABEL is the decision table. A decision table is a two-dimensional programming statement. While the English-like syntax of RAND-ABEL typically leads to verbose programs, the row/column tabular syntax of decision tables allows large quantities of data and expressions to be given in a compacted and readable form. Decision tables are used for expressing such things as decision logic and variable initialization. The contribution of decision tables to program clarity should not be underestimated.

5.6 ERIC

ERIC (Hilton, 1987) is an object-oriented simulation system developed by the Rome Air Development Center (RADC) in response to difficulties encountered in using ROSS. ERIC is essentially a rewrite of ROSS, using the OOP semantics of the Zetalisp Flavors system (Weinreb and Moon, 1980) rather than that of Hewitt's ACTORS system.

Objects in ERIC are arranged in a multiple inheritance hierarchy using a conflict resolution scheme similar to that found in Flavors. Like ROSS, objects have *attributes* and *behaviors*, and behaviors are activated via message passing in much the same pattern-matching manner. Like Flavors, ERIC distinguishes between *class objects* and *instance objects*; unlike ROSS, it does not treat class objects as prototypical instance objects. Instance objects can only have a single parent and may only inherit their behaviors. Other influences from Flavors include *before*, *after*, and *wrapper* daemons.

Like ROSS, ERIC only supports discrete-event deterministic simulation and models the advance of time with a predefined clock object. ERIC uses an event-scheduling approach, where the time is advanced to the time of the next scheduled event, which is then executed. ERIC also uses an efficient scheduling mechanism, which distributes the event queue between the objects that schedule events and, thus, reduces the size of individual event queues as well as the time needed to insert and remove events.

5.7 SIMKIT

SimKit (IntelliCorp, 1985b) is a package developed for use with the KEE (IntelliCorp, 1985a) system, a frame-based expert system development tool. KEE provides the underpinnings for the SimKit package. Entities, attributes, and relations are modeled with *frames* and *slots*—grossly stated, frames are an AI-based variation of objects. See Minsky (1975) for more details on frames.

Modeling entity behavior is facilitated by KEE's object-oriented programming capability. Particular sequences of behavior are coded in LISP and initiated through message passing. The KEE Rulesystem supports description of behavior in terms of forward-chaining rules as described in Section 4.1.

SimKit uses an event-scheduling approach to discrete-event simulation. It supports facilities for modeling stochastic variables and gathering statistical data during the execution of the simulation. SimKit is intended to supply simulation support to expert systems developed in KEE.

5.8 MODSIM

ModSim (West and Mullarney, 1987) is an object-oriented extension to Modula-2 developed specifically for discrete-event simulation. At the time of this writing, ModSim is still only a proposed system.

In the proposal for ModSim, concepts of OOP are added to Modula-2 as an appendage rather than smoothly integrated into the design of that language. The object-oriented component of ModSim conforms to the conventional view of OOP. Objects are arranged in a multiple inheritance hierarchy and object behavior is controlled via message passing. There is a separate syntax and semantics covering data structures represented using ModSim objects and those represented using standard Modula-2 data abstraction facilities.

With respect to simulation, ModSim is process-oriented. It seems to borrow heavily from concepts developed for Simscript II.5. However, while ModSim and Simscript share a common view of simulation programming, ModSim excludes Simscript's many eclectic features.

5.9 ROBS

ROBS (Radiya and Sargent, 1987) is a proposed language for discrete-event simulation that combines object-oriented and rule-based programming.

A ROBS object consists of *attributes* and *processes*. Unlike other object systems, the attributes of an object can actually belong to other objects; i.e., an object can have direct read-access to the local memories of other objects. Processes are the procedural aspect of an object and are activated via message passing. Unlike other object systems, processes are time-elapsing and allow ROBS to support a process-oriented world view.

A rule consists of a *condition* and a *consequent*. The condition of a rule is specified in terms of simulation time and the attributes of objects; the consequent is a sequence of actions, such as scheduling events or sending messages. Unlike processes, which are local to an object, rules are global. A rule is said to be *activated* when its condition is satisfied by system state.

A model specified in ROBS consists of an *initial situation*, objects, rules, and a *terminal condition*. The initial situation is a set of messages initializing the attributes of objects and scheduling activation of object processes; the terminal condition specifies when the run of a simulation is completed.

5.10 CLOSING COMMENTS

As Kiviat pointed out some twenty years ago, cogent arguments exist for *not* using simulation programming languages (Kiviat, 1969). Largely, these arguments are still valid. For instance, technical objections dwell mostly on poor execution speed, dubious reliability, and minimal programming support facilities, while operational objections focus on inadequate documentation and nontransferability across computing environments. Despite the validity of these objections, their edge of truth is thin.

Twenty years ago, simulation programming languages were in their infancy; perhaps now it is fair to say they have only reached adolescence and are slowly maturing with advances in simulation research. Then as now, many simulation languages emerge as the outgrowth of research projects, where efficiency and reliability are often secondary concerns. While some languages, such as GPSS, Simscript, SLAM, and a few others, are distributed by commercial vendors who are able to provide adequate technical support, other languages, such as ROSS, RAND-ABEL, and ERIC, are released to the public more as a convenience and intellectual gesture than as a profitable business venture.

Regardless of the noted drawbacks, there are still two good reasons for using simulation programming languages, namely, (1) programming convenience, and (2) concept articulation. A simulation programming language essentially provides a framework within which system models can be formalized. Such a framework focuses the efforts of the model builder, eliminates superfluous programming tasks and protects a naive programmer from reinventing modeling concepts on his own.

6. PERSPECTIVES ON MILITARY SIMULATION

To identify the design criteria that should guide the development of SERAS, we reviewed current and potential simulation technology, focusing primarily on that employed within RAND. This included:

1. A review of simulation techniques and languages;
2. A survey of contemporary military simulation models;
3. Interviews with simulation practitioners;
4. A review of advanced simulation research projects.

Our review of techniques and languages included contemporary systems for simulation—such as those developed at RAND (Simscript, ROSS, and RAND-ABEL), the Rome Air Development Center's (RADC) ERIC, and IntelliCorp's SimKit—as well as proposed systems—such as CACI's ModSim and Syracuse University's ROBS. Models surveyed included those developed at RAND, such as TAC SAGE, and TSAR, and those developed elsewhere, such as JANUS and IDAHEX. Our interviews were conducted with ten RAND researchers having varied experiences with and perspectives on the development and application of simulations. The RAND advanced simulation projects reviewed included Knowledge-Based Simulation (KBSim), RAND Strategy Assessment Center (RSAC), Concurrent Processing for Advanced Simulation (CPAS), and RAND Integrated Simulation Environment (RISE). Projects reviewed outside of RAND include those at RADC, MITRE, the Lawrence Livermore National Laboratories, and Hughes.

In the three preceding sections, we presented the results of our review of simulation programming technology. Here, we summarize our review of military simulation projects and perspectives, largely drawn from RAND. This is not intended to be construed as an in-depth study. It was conducted informally as a means of providing direction to the design and development of SERAS.

6.1 ASPECTS OF MILITARY SIMULATION

Simulation existed as a powerful tool of the military long before the advent of computers, and the potential of computer simulation has been explored extensively since. Military simulations range from economic and political models of international peace and war to

representations of individual air and ground battles. Despite the level of effort applied to military simulation, existing technology remains archaic. Due to security reasons, simulation studies of military and defense systems are largely unpublished, which hinders the exchange (and, thus, the advance) of military simulation technology. Several aspects of military systems make them less tractable to contemporary simulation techniques than systems of interest to industrial, economic, and social studies.

Military and defense systems are more complex than systems in most other domains. All military systems share elements of strategy, command and control, tactics, logistics, intelligence, and communications, yet knowing this does not make these concepts easier to instantiate, factor, or model. In addition, contemporary simulation programming languages provide minimal support at best for these concepts, which may account for the fact that many military simulations are still implemented in general-purpose languages such as Fortran.

Military systems are dynamic and stochastic. Strategy and tactics change. Forces and weapons systems are deployed and utilized, reduced through attrition, regrouped and redeployed. Communications can be jammed and otherwise disrupted. Weapon systems can fail. Intelligence can be uncertain or erroneous. Military systems also exist in an open environment and must deal with numerous exogenous factors.

Yet another problem with military systems is that typical entities, such as command posts, forces, weapon systems, etc., often rely upon human judgment and decision-making capabilities. Since the modeling of human cognition is still an unsolved problem and an active area of research in artificial intelligence and cognitive psychology, modeling it using contemporary simulation technology is problematic.

Finally, military systems are data rich. National command level systems must literally deal with an entire world of data. Theater and tactical systems must deal with large sets of geographic and geopolitical data. These data are not even static, changing according to the effects of conflicts. In addition, military systems must contend with weather data and data along the electromagnetic spectrum, whose dynamics are generally exogenous. Contemporary simulation techniques do not provide general mechanisms for representing the varying aspects of this vast amount of information or working with it in a reliable manner.

6.2 OVERVIEW OF MILITARY SIMULATION MODELS

In our survey of existing military simulation models, we looked at eight systems, developed both here at RAND and elsewhere. Included in our survey are TACSAGE/CAMPAIGN, S-Land, IDAHEX, JANUS, VIC, SIMSTAR, STOMPEM, TSAR and TSARINA, and ESAMS.

6.2.1 TACSAGE/CAMPAIGN

TACSAGE, developed at RAND in Fortran, is an air-land warfare "optimization" computer simulator. The SAGE algorithm incorporates a trial-and-error search for finding an "optimal" solution to a linear approximation of a two-sided resource-allocation game, using the simulation as a method for rating potential allocation schemes. The land-combat simulation to which TACSAGE is presently attached is the CAMPAIGN simulation. CAMPAIGN was developed in the C language as the main theater model for the RAND Strategy Assessment System (RSAS); it was modified for use with TACSAGE. CAMPAIGN is a theater-level simulation of air-land combat, including attrition, movement rates, and a limited but developing representation of maneuver warfare.

6.2.2 S-Land

The S-Land (secondary land) theater model was originally developed for the RSAS as a simulation of all air-land theaters except Central Europe and Korea. Written in the RAND-ABEL language, it makes extensive use of the ABEL decision table structure for both flexibility and transparency. The S-Land model has been modified for use in the RAND-ABEL Modeling Platform (RAMP) as a stand-alone development tool for all air-land theaters of operation. Use of RAND-ABEL's selective interpreter allows models to be developed interactively and later compiled for efficient production.

6.2.3 IDAHEX

IDAHEX is a theater-level air-land battle war game developed at the Institute for Defense Analysis (IDA); both the SHAPE Technical Center and RAND have modified the version of IDAHEX used at RAND. IDAHEX is a two-sided simulation played interactively by Red and Blue players, along with a controller (optional), on separate, distributed computers. It encompasses the essentials of maneuver over a hexagonal representation of the playing surface. Air combat is

optionally supported by an associated module (ACELAWS) incorporated into the code and data structures of the original IDAHEX land battle simulator.

6.2.4 JANUS

JANUS, developed at Lawrence Livermore National Laboratories (LLNL) and at U.S. Army TRADOC Systems Analysis Activity (TRASANA), is a Fortran-based combat simulator for forces at the division level and below. The modular framework provides for the modeling of a variety of weapon systems and other components to a high level of detail. Analytic results are augmented by high-resolution graphics that include maneuver and three-dimensional terrain with line-of-sight calculations.

6.2.5 VIC

VIC (Vector In Commander) is the Army's corps-level air-land battle model being developed at TRANSANA. The model is written in Simscript II.5 as an event-oriented, deterministic, expected-value, two-sided explicit representation of maneuver. Units are resolved to the level of battalion for the Blue side and regiment for the Red side. Lanchester battle outcomes are used. The effects of weather, terrain, and time of day have an impact on visibility and trafficability. Major force processes and components in the model include: maneuver unit combat; a top-down hierarchy of command, control, and communications (C^3); engineer operations; target acquisition and intelligence; support fire allocation; tactical air operations for close through deep battles; air defense; and a logistics supply network.

6.2.6 Other Models

Other analytic models include SIMSTAR, STOMPEM, TSAR/TSARINA, and ESAMS. SIMSTAR is a detailed communications model for analyzing the resilience of a variety of user-specified communications networks to direct and indirect degradation. STOMPEM is a "corridor" aircraft penetrator/attrition model that includes the major elements of electronic warfare (EW). TSAR is a detailed sortie generation simulation and TSARINA is a detailed air base attack model, outputs from which can be redirected to TSAR, affecting the sortie generation process. ESAMS is a physics model for analyzing one-on-one aircraft and surface-to-surface missile engagements.

6.3 SIMULATION INTERVIEWS

To get a first-hand understanding of simulation technology at RAND, a set of interviews was conducted informally with ten simulation practitioners, all having different levels of simulation experience, different views on building simulations, and different sets of expectations for simulation. The interviews did not follow a predefined format. Each interviewee was asked to discuss his experience with simulation and his views on the nature of simulation and simulation programming. If the interviewee's experience was with one particular simulation system, then he was asked to describe that system and the techniques used to build it.

The ten interviewees were:

1. Patrick Allen, who coauthored the S-Land theater model, as well as modified and used the IDAHEX and CAMPAIGN models.
2. Barry Wilson, who works on Analytic War Plan development as part of the RSAS and also coauthored the S-Land model.
3. James Bigelow, who has developed a small-scale reproduction of a NATO-theater simulation.
4. Joseph Bolten, who has developed numerous small- to medium-size simulations.
5. Edison Cesar, who served as domain expert for the ROSS simulation, TWIRL.
6. Carl Jones, who is working on the Campaign Model of the RSAC.
7. Donald Emerson, who developed the TSAR and TSARINA simulations.
8. Richard Hillestad, who developed the TACSAGE simulation.
9. Louis Moore, who has built many Fortran-based combat simulations.
10. Warren Walker, who has developed several large-scale operational simulations.

Most simulation programmers interviewed were satisfied to remain within the strictures of the language and techniques with which they were most familiar; typically, Fortran, for reasons of execution-time efficiency. Those who had gone beyond Fortran (e.g., to C and RAND-ABEL) seem to appreciate that there might be better languages. Likewise, there was agreement from those who developed long-lived simulations (e.g., TSAR and TACSAGE) that Fortran simulation programs were difficult to maintain and modify and that they were thinking of moving to another language (except for the

tremendous effort it would take). In all, there was not a general antagonism toward simulation-oriented programming languages. Rather, the opinion was that such languages (1) required learning new programming styles, (2) traded flexibility for efficiency, and (3) provided little support particular to military simulation. This indicates that to gain acceptance, developers of languages for military simulation should demonstrate how to use these languages to their full advantage on realistic systems. Further, during the early stages of language distribution, developers should provide personal instruction on using their language, not just reference manuals and user guides.

The concern about execution efficiency came up consistently in each interview. In many cases, it was either compile-time or execution-time performance that caused Fortran to win out over Simscript (despite the many facilities that Simscript provides for simulation). Given this, any new language must have performance as one of its primary objectives. The problem is that for advanced, high-level languages, there is often a trade-off between compile-time efficiency and the quality of the object code produced. To improve the object code, the compiler must necessarily spend more time on analysis and optimization. This means the compiler will take longer to run, which will hamper development time. Given a slow but powerful compiler, the way to improve development efficiency is to provide either an interpreter or a fast non-optimizing compiler or both. This would allow users to rapidly test and evaluate simulation programs. In addition, because of the interactive nature of an interpreter, it naturally supports debuggers, tracers, and other software development aids. Once a simulation program was developed, then it could be passed through the "super-compiler" to produce an efficient object program for fast simulation runs. Although this scheme could double the effort required by the developers of simulation languages, it would make their languages a viable alternative to Fortran.

Modeling issues that were considered problematic, particularly for object-oriented simulation, included representations of terrain, transportation and communication networks, geographic and geopolitical boundaries, and weather conditions. Existing military simulations typically use hexagons or rectangular "zones" for terrain representation. (Piston models are generally used for representing forward troop movements, hex models for representing terrain features.) There are also several hybrid and *ad hoc* representations. For instance, the RSAC uses an approach of overlaying multiple styles of representation. Representations of such things as weather and communications were less well defined than those for terrain and typically appeared on "things-to-do" lists. We do not at this time have a proposal for dealing

with these representations. It suggests an area for further research and experimentation. Perhaps the best support that an object-oriented simulation language can offer is a flexible data-structuring mechanism. For instance, many object systems enforce a single underlying implementation of all objects; such a restriction would hinder experimentation. Additionally, all the aforementioned aspects of a system model are generally data rich, drawing from a large data base. This means that the language should also provide data-base access facilities. For efficiency, these facilities might operate by loading relevant sections of a data base "in-core" and "swapping" sections as the requirements of the system change.

The representation and use of military doctrine was an important issue to some analysts interviewed. Military doctrine essentially defines the behavior and constraints on the entities of military systems. Simulations could be used to teach and demonstrate military doctrine as well as to evaluate it. The problem is that simulation programs are rarely coded in terms of military doctrine, and there are no standard techniques for integrating doctrine into a program. This makes experimentation, validation, and modification complex and time-consuming at best. In addition, doctrine changes frequently, requiring constant maintenance of simulation programs. A language (or development environment) for military simulation should recognize the importance of doctrine, possibly providing facilities that take doctrine as input and automatically integrate it into a simulation program.

Other requirements for military simulation that came out of the interviews include:

- Support for implementing modules in a lower-level language and facilities for creating external interfaces that smoothly integrate such modules into the simulation program;
- Support for continuous valued variables within a discrete-event simulation language;
- Support for both quantitative and analytic processing, and qualitative and inferential processing;
- Support for general-algorithmic (ALGOL-like) processing;
- Facilities for selectively gathering simulation results;
- Extended data collection, analysis, and display facilities that aid in distinguishing differences between simulation runs.

The above discussion reflects only a brief synopsis of comments and ideas drawn from the interviews. The interviews provided a starting point for developing our ideas about the nature of SERAS. More details on the design criteria we finally developed can be found in Section 7.2.

6.4 REVIEW OF ADVANCED SIMULATION RESEARCH

The RAND Corporation has a long history of conducting innovative research in the development of computer simulations. Simulation research at RAND produced Simscript I (Markowitz, Hausner, and Karr, 1962) and Simscript II (Kiviat, Villanueva, and Markowitz, 1968), as well as theoretical and experimental results in game theory, Monte Carlo simulation, and military war gaming (Conway, 1962; Kamins, 1963; Ginsberg, Markowitz, and Oldfather, 1965; Sharpe, 1965; and Voosen, 1967). RAND currently has several active projects addressing advanced techniques in using and building computer simulations. Those that we focused on in developing our ideas about the needs of advanced simulation research include:

- The RAND Strategy Assessment Center
- The RAND Integrated Simulation Environment
- Concurrent Processing for Advanced Simulation
- Knowledge-Based Simulation

6.4.1 The RAND Strategy Assessment Center

The RAND Strategy Assessment Center (Davis and Winnefeld, 1983) is a large-scale DoD project to develop new concepts and techniques combining features of war gaming and analytic modeling. The centerpiece of the project is a computerized war gaming system, the RAND Strategy Assessment System (Davis, Bankes, and Kahan, 1986), in which some or all political and military national decisions can be made programmatically, and in which both force operations and combat are described by theater- and strategic-level models. Essentially, RSAS is a simulation/war gaming environment for global multi-theater conflicts.

The RSAS combines aspects of knowledge-based modeling with conventional simulation techniques. It consists of four major components, Blue Agents, Red Agents, a Scenario Agent, and a Force Model. The Blue and Red Agents model the decision-making elements of the NATO alliance and the Eastern bloc, respectively. The Scenario Agent models other decision-making elements, such as those of third world countries. The Force Model consists of a global conflict simulation, i.e., it maintains the game clock, moves and keeps track of military forces, assesses the results of battle, and generally instantiates the decisions of the three decision-making agents.

A current effort in the RSAC is under way to organize and refine the software underlying the RSAC models and create a generic modeling environment. This software and associated documentation,

preliminarily named the RAND-ABEL Modeling Platform (RAMP), will include the RAND-ABEL decision modeling language, as well as control software for use in developing multi-player models and several sophisticated interface tools for the display and entry of simulation data through tabular and graphical devices.

6.4.2 The RAND Integrated Simulation Environment and Concurrent Processing for Advanced Simulation

The RAND Integrated Simulation Environment and the Concurrent Processing for Advanced Simulation projects are loosely connected with the objective of developing a simulation programming environment that can take advantage of distributed computing. CPAS is rooted in earlier RAND work in distributed computation, Time Warp (Jefferson and Sowizral, 1982). RISE is an object-oriented simulation programming environment for land battle simulations and is intended to make use of advances in distributed computing technology that result from CPAS.

6.4.3 Knowledge-Based Simulation

The Knowledge-Based Simulation Project (Rothenberg et al., forthcoming) followed the ROSS language development effort. The objective of this project is to demonstrate the utility of AI techniques to help solve deficiencies that exist in large-scale military simulations. The project synthesizes ideas and techniques from AI and expert system technology and from graphics.

The research agenda for KBSim is divided between two broad areas: extending the object-oriented modeling paradigm; and intelligent explanation and exploration.

Extended Modeling Paradigm. This task focuses on adding a richer set of modeling concepts to object-oriented simulation. KBSim is exploring the use of inference and condition-action rules to represent the behaviors, intentions, and reasoning processes of objects as well as constraints on object behavior and the behavior of the simulation as a whole. Similarly, KBSim is experimenting with declarative and demand-driven programming (Cammarata, Gates, and Rothenberg, 1988) to automate certain aspects of simulation programming such as unplanning, control of inference, control of aggregation/disaggregation, and graphic presentation and interaction.

Intelligent Explanation and Exploration. This task focuses on developing techniques for making the results of a simulation, as well as the information encoded therein, more readily understandable and

accessible. Explanation requires that a simulation keep track of pertinent information regarding its execution and display that information in a meaningful way. Exploration requires allowing analysts to selectively modify a simulation, pursue excursions, focus attention on selected aspects of the model, perform sensitivity analysis, or ask how particular results might be achieved.

7. SUPPORTING TECHNOLOGY TRANSFER

Up to this point we have presented a compilation of our review into simulation concepts and technology. In this section, we return to considerations of technology transfer, outlining a possible design of the SERAS programming system.

As mentioned earlier, the purpose of the Transfer of Simulation Technology Project is *not* to build the better simulation language. Rather, its purpose is to develop a system that aids in porting knowledge-based and object-oriented techniques in military simulation to programming environments that analysts will be willing to use and evaluate on realistic models. Such a system is intended to help make the results of advanced simulation research effectively accessible to military analysts, bridging the language gap between research and applications and, in this way, supporting technology transfer.

Our own view of SERAS is that it provides a conceptual framework in which to explore the space of possible simulation programming languages. By supporting high-level mechanisms for control and data abstraction, SERAS provides researchers with tools for developing new approaches to simulation programming. By supporting mechanisms for mapping syntax to semantics, SERAS enables researchers to rapidly prototype new simulation languages in which to demonstrate and evaluate their ideas.

Because we do not wish to overtly bias users, SERAS does not espouse any single approach to simulation programming. However, to aid in the exploration of new ideas, we believe that SERAS should include a baseline simulation language that users can extend, enhance, or otherwise modify. The baseline described in this section draws upon features found in existing simulation languages that seem suitable to military simulation, other features not normally found in simulation languages but that the SERAS control abstractions make feasible, and new features that have been developed by current RAND research.

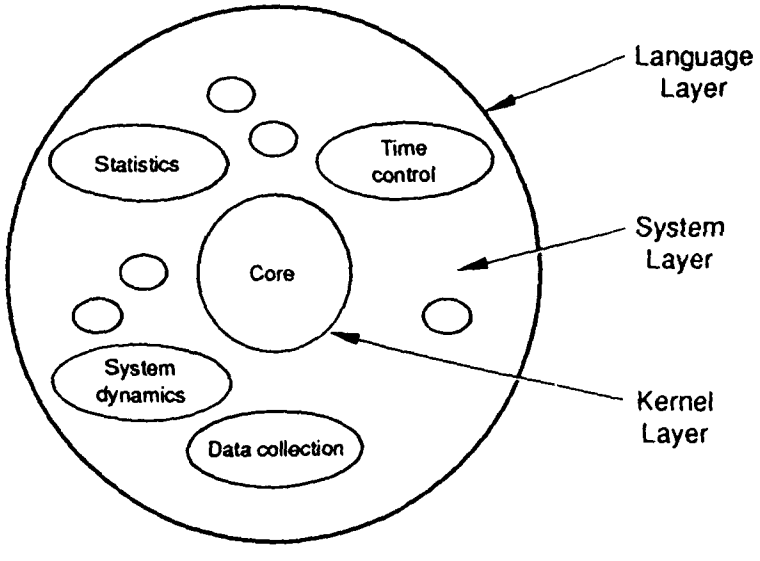
7.1 A FRAMEWORK FOR LANGUAGE EXPLORATION

The SERAS programming system consists of two component programming languages: *Core* and *SLAG*. Taken together, these languages enable the rapid prototyping of advanced simulation languages, which can be used to demonstrate and evaluate new approaches in simulation programming.

The generic term for a SERAS prototype . SL_i. The anatomy of an SL_i can be viewed as being organized among three layers. At the surface, we have the *Language Layer*. Underlying this is the *System Layer*, and at its heart is the *Kernel Layer*. The figure seen below illustrates the relationship of these three layers with a cross section of an SL_i.

The Language Layer defines the syntax of the SL_i. This layer is oriented toward military analysts for use in encoding simulation models. It allows analysts to view each SL_i as a specialized, high-level modeling language. The Language Layer is defined with a grammar that maps an abstract, model-oriented syntax into semantic constructs of the underlying System Layer. This grammar is then input to the SLAG component of SERAS. SLAG (Simulation LAnguage Generator) uses this grammar to construct a compiler and interpreter for the SL_i. Military analysts can use the interpreter to develop system models and, later, compile their models for efficient performance when running simulation experiments.

The System Layer underlies the Language Layer and defines the semantics of the SL_i. This layer consists of a collection of software packages that support simulation programming, such as abstractions for modeling static and dynamic system structures, mechanisms for



Anatomy of SL_i

modeling stochastic aspects of a system, and facilities for data collection and analysis. Modules within this layer can also provide advanced simulation functionality, such as support for dynamically changing the granularity of a model, support for describing complex relationships among entities, and support for representing problematic aspects of object-oriented simulation, such as terrain features, weather conditions, and transportation networks. The System Layer is expected to be customized by researchers to formalize and demonstrate their ideas and techniques. It provides a platform upon which to experiment with new concepts in simulation programming.

The Kernel Layer is the common base upon which all SL_i are built. It consists wholly of the Core system development language. The intended use of Core is in the definition of the System Layer. Thus, Core is not a simulation programming language but a language for defining the semantics of simulation. In our proposed design of Core, we integrate concepts from AI, such as rule-based processing, backward- and forward-inference, and resolution and unification, with concepts from OOP, such as data abstraction, encapsulation, polymorphism, and inheritance. We feel that such a multiparadigm design will provide the basic processing requirements needed for developing a diverse set of software packages for advanced simulation programming.

In combination, these three layers form a prototype simulation programming language generated with SERAS. New simulation technology can be made accessible to military analysts by embedding it in such prototypes. In the remainder of this section, we outline the design of Core, we present the semantics for a baseline prototype simulation language (SL_0), and we give an overview of SLAG.

7.2 A DESIGN OF CORE

Above we characterize Core as a system development language. Our intention is that simulation researchers will use Core to develop knowledge-based and object-oriented simulation software packages, which they can then embed in a prototype simulation language SL_i and make available to military analysts for demonstration and evaluation on realistic models. Thus, the design of Core must be oriented toward this task and not the task of encoding system models. To support such an orientation, we propose using a multiparadigm design for Core, which integrates concepts and technology from procedural programming with those from rule-based and object-oriented programming.

7.2.1 Design Criteria

The motivation for our design of Core comes from our review of simulation technology, applications, and practices. While this review was conducted informally, it provided us with sufficient insight into the needs of simulation programming to develop a list of features and capabilities for which Core should provide some form of support. What follows is a brief discussion of some of these capabilities and their role in simulation programming.

- *Abstraction* and *Encapsulation*—Simulation programs must be maintained and kept up to date; they share many of the same component subsystems; and they must be available to a variety of computing environments. The key to ease of maintenance, reusability, and portability is implementation independence, i.e., separating application from implementation. Implementation independence can be supported with mechanisms for abstraction and encapsulation, where we view abstraction as *allowing* one to talk about programming concepts independent of their implementation and encapsulation as essentially *forcing* one to talk in this manner.
- *Associative Data Processing*—Simulation programs must be able to represent and work with relations among entities in a system model. Relations can easily be represented with any of a variety of associative data structures, such as lists, arrays, or tables. However, they still must be thought of and worked with in the abstract. As a result, associative data processing does not merely mean providing operations on associative data structures; rather, it means providing a mechanism for searching the abstract space of relations.
- *State-Based Processing*—In general, there are two ways in which state-based processing enters into a simulation program. First, simulations monitor the behavior of entities in a system model and collect statistical data according to changes in the state of entities over time. Second, while in normal situations the behavior of entities can be based upon predetermined courses of action, entities must be able to recognize exceptional situations and change their behavior accordingly. To support both tasks, simulation programs require mechanisms for monitoring system state and for initiating the appropriate actions when the system enters particular state configurations.
- *Concurrent Processing*—The behavior of an entity can be thought of as modeling how it undertakes an activity, and an arbitrary number of entities can be undertaking distinct

activities during the same period of simulation time, i.e., concurrently. This capability requires mechanisms for representing concurrent activities as separate processes, for moving each process ahead in time, and for switching control between processes. In addition, these mechanisms can be generalized to support peripheral processes that go on concurrently with a simulation, such as those that collect statistical data or those that interface to a graphics device.

- *General Algorithmic Processing*—While simulation programs require the special processing capabilities indicated above, many tasks simply require conventional programming utility, i.e., features such as subroutines and recursion, structured iteration, and so forth. Thus, support for these programming features must not be given less consideration than other more technically advanced or interesting features.

Of course, this is not an exhaustive list; additional capabilities include statistical data collection, analytic and arithmetic processing, output formatting and generation, foreign-function calling, interfacing to data bases and graphical devices, as well as others. We have highlighted the capabilities seen above, because they were instrumental in motivating our design of Core as a language that integrates concepts from procedural, rule-based, and object-oriented programming—we use the term *state-based programming* to characterize forward-chaining rule processing and *logic programming* to characterize backward-chaining rule processing.

Beyond support for the capabilities enumerated above, the single most significant design criterion for Core is that of execution efficiency. Unless the prototype simulation languages based on Core can be used to model large military systems with reliable performance, military analysts will treat them as toy languages, i.e., cute but of no practical value. While Core is intended to have the general appearance of LISP, if execution efficiency is to be achieved it cannot be a dialect or add-on to LISP. Core should be a typed language. Although not strongly typed (i.e., type checking that cannot be done at compile time must be deferred to execution time), Core should be closer to Ada in regard to typing than LISP. Core should also be a compiled language for which an interpreter exists, as opposed to LISP, which is an interpreted language for which compilers exist. This means that there is a strong emphasis on distinguishing between compile-time and execute-time computations. Finally, in LISP, programmers often off-load their responsibility for proper memory management to the garbage collector, which is incapable of doing this job with optimal or even near-optimal performance. As the design of Core matures, restrictions should be

introduced on the allocation and reclamation of memory in an attempt to enforce programmer responsibility for the efficient use of space. This does not necessarily mean eliminating the garbage collector; instead, it could simply mean making the programmer cognizant of memory usage and providing mechanisms for its optimization.

While it can be argued that support for these capabilities can be implemented on top of an existing simulation language, such as ROSS, this argument ignores two properties that are essential to the effective utility of Core, namely, coherence and efficiency. To smoothly integrate the techniques that support these capabilities they must be made a fundamental part of Core's design, not add-ons; and, to provide efficient execution-time performance, the Core compiler must also be able to treat these capabilities as primitives over which it can optimize for efficiency.

7.2.2 Programming Concepts

Below we outline the design of Core as a programming language. The discussion that follows is intended to provide a high-level overview of its syntax and semantics. In particular, we examine aspects of the design motivated to support the capabilities presented in Section 7.2.1. (An in-depth discussion of these topics is outside the scope of this document.)

Generally, the syntax and operations of Core borrow from Common LISP (Steele, 1984). There are several reasons for this.

1. Each prototype simulation language consists of a translator for converting programming constructs of the prototype into their semantic equivalent in Core code. Using a language with the simple syntax of LISP for the target eases the task of defining the code generation rules for the translator.
2. The target user group for Core consists largely of researchers with extensive LISP experience. By mimicking LISP, we hope to ease their transition into using Core.
3. LISP is a system development language. Common LISP provides standards of support for many of the auxiliary capabilities that we plan to support in Core. By structuring Core after Common LISP, we can readily use Common LISP for the initial implementation of Core and tie into this existing support, speeding up development of a demonstrable prototype.

Regardless of our plan to borrow from Common LISP, Core should not be thought of as an "add-on" or extension of Common LISP. We feel

that the dynamic semantics of LISP seriously hinder efficient execution-time performance. As a result, the semantics of Core will differ significantly from LISP in terms of its static and dynamic structure.

Syntax. The basic lexical and syntactic rules of Core are very simple. Core uses a fully parenthesized Cambridge Polish notation and, thus, has the general appearance of LISP. Lexical elements are delineated by nonprinting characters such as spaces, tabs, and end-of-lines, while complex syntactic forms are delineated by matching left and right parentheses. The fundamental lexical categories include *numbers*, *strings*, *symbols*, and *keywords*. Parenthesized forms, when discussed in terms of their structure, are called *lists*.

Unlike LISP, where all lists can be treated both as literals and as expressions with values, the semantics of lists in Core are much more rigidly defined and largely determined by their lexical orientation in the program. For instance, if a list is a top-level element in a program block, then it is called a *statement*; if it appears in an argument position, then it is called an *expression*; if it is used to compute a Boolean value, it is called a *predicate*; and so forth.

Data Abstraction and Encapsulation. Core supports abstraction and encapsulation using concepts originating from OOP and influenced by mechanisms found in the Common Loops Object System (Bobrow et al., 1988) as well as Ada (U.S. DoD, 1983) and the LISP-like languages Scheme (Abelson and Sussman, 1984) and T (Rees, Norman, and Meehan, 1983).

Following conventional OOP terminology, all data elements of Core are called *objects*, and each object is an *instance* of a class. Portions of the definition of a class can be *inherited* from other classes, called *superclasses* of the class. Additionally, each class is an object and, therefore, an instance of a class; a class whose instances are classes is called a *metaclass*. A metaclass defines the behavior of its instance classes. For example, a metaclass defines how the semantics of inheritance operates, how instances of the class are generated, and how instances of the class relate to other objects.

Although the exact semantics of a class is dependent upon the metaclass of which the class is an instance, in general, when a programmer defines a class he is defining an abstract data structure. A class allows him to designate (1) how instances of the data structure are implemented, and (2) what primitive operations are needed for working with that implementation.

Operationally, a class couples a mechanism for encapsulating the constructor and other primitive operations of the data structure with a

mechanism for relating instances of the data structure to other objects in an implementation-independent manner. In Core, we support these mechanisms with two lower-level programming constructs. One, called *packages*, provides for general encapsulation; the other, called *types*, provides for data abstraction.

Packages in our design are similar to the concept of packages found in Ada; i.e., a package delineates the lexical scope of some body of code and provides an external interface that allows users of the package to access particular portions of the code otherwise hidden within the scope of the package. Types, on the other hand, are modeled after the concept of types found in Common LISP—they can also be viewed as integrating Ada's concepts of types, subtypes, and generics. Essentially, a type delineates an abstract grouping of objects, independent of their implementation. Programming constructs that return values, such as function calls and variables, are defined in terms of the types of objects they return; likewise, parameterized programming constructs, such as subroutines, are defined in terms of the types of arguments they accept. New types can be created by either merging subtypes into a large, more general supertype, or by partitioning a supertype into several smaller, more specific subtypes.

There are several reasons for supporting packages and types as programming constructs separate from classes. In particular, one might wish to encapsulate over a set of operations, even though those operations do not correspond to a data structure. Likewise, one might wish to join or partition groups of objects without defining a new data structure. Providing packages and types, respectively, supports these needs without otherwise overloading the class concept. Additionally, the ability to define new metaclasses necessitates that programmers be given hooks into the fundamental components of a class.

Generic Operations and Methods. Subprograms, i.e., procedures and functions, are defined as *generic operations*. A generic operation can have an arbitrary number of definitions. Each definition, called a *method*, designates the number and types of its formal parameters and the body of code to be executed when applied.

When calling a generic operation, the number and types of the actual parameters in the calling form are compared to formal parameters of the methods defined for that operation. The method whose parameters have the "best" match is selected as the appropriate method to apply.

In the case where more than one method is applicable to a given calling form, the "best" method is considered to be the most specific. Unfortunately, specificity is not a scalar metric. When two or more methods are equally specific, the user must provide some other means

for disambiguation—for instance, the user could define an explicit ordering on methods (e.g., by recency, priority, or by left-to-right parameter comparison)—otherwise, an ambiguity warning is triggered.

Concurrent Processing. As in SIMULA, a method can be thought of as a class object; i.e., a method is a formal description of a process with local memory, algorithms, and actions. When applied, a dynamic instance of the method is created, which we call a *process instance.*

Normally, when a call to a generic operation is made, a process instance is created, allowed to run and terminate, and then discarded. The **make-process** operation, which has the form

(make-process
:form { *procedure call* | *function call* | *proposition* }**)**

can be used to get a handle on a process instance and run it as a coroutine. The **make-process** operation creates a process instance, allows it to run some initialization code, and then returns a pointer to the process.

A process instance can be in one of three states, *active*, *detached*, or *terminated.* It is active when executing the statements in its body, detached when it has executed the **(detach)** statement, and terminated when it returns control by any other means. A process instance returned by **make-process** is initially detached. If the form used to create the process is a function call or a proposition, then the process is called a *stream* and expected to produce values upon detaching.

A detached process can become active again with the **resume** operation. The statement

(resume *process***)**

will resume computations of *process* beginning with the statement following the last call to **detach**, assuming *process* is in the detached state; if *process* is terminated, then **resume** returns immediately.

Associative Data Processing. Associative data processing is supported with a form of logic programming, the constructs of which we call *relations*, *facts*, *propositions*, and *well-formed formulas.* A relation describes an association that can exist between particular types of objects. A fact is an instance of a relation involving actual objects of those types. A proposition tests the truth or falsity of a fact, while a well-formed formula (*wff*) is a rule for proving the truth or falsity of a proposition using resolution and unification.

For example, the statement

```
(defreln father (child father)
 (declare
  (variable child person)
  (variable father man)))
```

defines that a **father** relation can hold between objects of type **person** and **man**. The form

```
(make-fact father
 :child mary
 :father john)
```

creates an instance of this relation, i.e., a fact that states that **john** is the **father** of **mary** (assuming **john** belongs to type **man** and **mary** to **person**. (This can be abbreviated to **#F:(father mary john)**.) The statement

```
(if (father mary john) . . . )
```

shows a proposition that tests whether **(father mary john)** holds. Finally, the statement

```
(defwff father (child father)
 (declare
  (variable child person)
  (variable father man)
  (variable mother woman))
 (and (mother ?child ?mother)
 nau'(married ?father ?mother)))
```

defines a wff for proving that an instance of the **father** relation holds, i.e., as a logical conjunction of two propositions. (A question mark (?) is used to indicate call-by-reference.)

State-Based Processing. Core supports state-based processing with constructs called *switches*, *rules*, *controllers*, and *rulesets*. A switch is a primitive programming structure for monitoring system state. Rules are defined in terms of switches. The firing of a rule is determined by a recognize-act cycle monitored by a controller. Rulesets are essentially rule subroutines.

A switch has the general form

```
(switch condition
 statement)
```

When *condition* is met by the current system state, an instantiation of the switch is created. The *statement* part is turned into a process

(particular to the instantiation) and allowed to run until it detaches. When the condition under which the instantiation was created no longer holds, then this process is resumed and, after it detaches again, the instantiation is discarded. Thus, one might think of a switch as a demon that is allowed to clean up after itself.

To illustrate this concept, consider the following switch

```
(switch (and (= (sensing-range ?aircraft) ?range)
             (< (distance ?aircraft ?radar) ?range))
 (let ((fact #F:(senses ?aircraft ?radar)))
  (assert fact)
  (detach)
  (retract fact)))
```

which reads as,

WHEN:
the sensing range of some **?aircraft** *is* **?range** *and*
the distance between **?aircraft** *and* **?radar** *is less than* **?range**,
THEN:
assert that **?aircraft** *senses* **?radar**;
WHEN:
that distance again becomes greater than **?range**,
THEN:
retract the assertion.

For efficiency, the condition of a switch is monitored using a variation on the RETE algorithm (discussed in Section 4.1).

Rules are essentially structured switches. However, where the activation of a switch is independent of other objects, the activation of a rule is dependent upon the controller to which the rule belongs. A controller defines an inference engine over a set of rules; an arbitrary number of controllers can be present concurrently. When defining a controller, one is generally describing how instances of the controller will behave, i.e., how they select rules to fire. When an instance of a controller is created, it is given a set of rules to monitor. While enabled, it operates concurrently with other active controllers, selecting and firing rules according to a recognize-act cycle.

The dynamic scope of a controller and its rules can be delineated by the use of rulesets. A ruleset is essentially a generic operation whose body consists of a controller and a set of rules, rather than a set of programming statements. When called, the controller of the ruleset is enabled and its rules allowed to fire. When the ruleset is exited, the controller is disabled.

7.2.3 Implementation Issues

To implement Core so that it contains the features outlined above, several issues must be addressed. Of these, issues of integration and usability are probably the most pressing.

In our design of Core, we have taken a *multiparadigm* approach (Bobrow, 1985); i.e., we combine concepts from procedural, rule-based, and object-oriented programming into a single language. We have pursued this approach because it best supports the task for which Core is intended. Each of the paradigms has its own unique strengths. Concepts from object-oriented programming support the organization and use of data and procedures, while concepts from procedure-oriented and rule-based programming support distinct modes of processing, each of which has utility to simulation programming. By combining these paradigms we can provide built-in support for a wider range of functionality than is possible by simply basing Core on just a single paradigm.

Methodologies for the design and implementation of multiparadigm languages are still an open research area (Hailpern, 1986). Integration (properly merging the paradigms) is one of the most significant problems. Even though we will be drawing upon existing technology, similar attempts have resulted in producing "kitchen sink" languages, i.e., languages that support a host of technically interesting but operationally incompatible components. So, care must be taken to ensure that our design is a smooth integration that forms a coherent whole. Briefly, we feel that the key to smooth integration is abstraction. It is not enough to simply provide a means for defining programs in terms of several different paradigms; users must also be able to access these paradigms transparently. For instance, if the user calls an operation, he should be able to do so without having to know that the operation is defined as a method, a ruleset, or a wff.

The usability issue involves conflicts between Core's development environment and its execution environment. If SERAS prototypes are to undergo realistic evaluation by military analysts, their execution speed must be comparable to languages with which the analysts are familiar. Since the execution-time performance of any prototype is based on the performance of Core, Core's execution environment must be as efficient as possible. Because of the complexity of Core's design, the best methods for obtaining optimal execution-time performance are (1) to make many aspects of the language static and (2) to give the compiler a great deal of information about Core programs.

The problems with these methods to improve execution-time efficiency would appear when trying to use Core as a software

development tool. Researchers using Core to explore new approaches to simulation programming are intended to view it much like LISP, i.e., in an interpreted programming environment. Such an environment allows them to develop and test code rapidly, without wasting time recompiling after every edit. However, very few aspects of LISP are static (e.g., even function definitions can change during the execution of a program). By making many aspects of Core static, many of the execution-time operations required by LISP can be compiled away, improving performance but possibly hampering software development. The second method to improving execution-time efficiency requires inserting numerous compile-time declarations into Core programs. This is a time-consuming and tedious task, and at the initial stages of development many declarations will not be known, so this too can hamper rapid system development. A possible approach to overcoming these conflicts is to provide a SERAS with a strict Core compiler, a forgiving Core interpreter, and tools for automating the process of adding declarations to code developed under the interpreter.

7.3 A BASELINE SEMANTICS

In this section, we outline the semantics of the baseline prototype language (SL_0) that we hope to include with SERAS. The System Layer of SL_0 will consist of a collection of simulation-specific software packages implemented with Core. SL_0 combines the event-processing, activity-scanning, and process-orientation world views and includes a form of continuous variables and functions.

As discussed in Section 3, there are four general areas of fundamental functionality that any simulation programming language should support:

1. Methods for describing the static structure of a system
2. Methods for describing system dynamics
3. Statistical sampling facilities
4. Facilities for data collection, analysis, and display

What follows is a brief discussion of how SL_0 provides this level of functionality. We also include a discussion of other "advanced" simulation constructs we hope to support, such as planning and unplanning, constraints, and dynamically varying levels of aggregation.

7.3.1 Modeling Static Structure

For describing the static structure of a system model, a simulation programming language must be able to

1. Define the classes of objects within the system
2. Adjust the number of these objects as conditions within the system change
3. Define attributes that can both describe and differentiate objects of the same class
4. Relate objects to one another and to a common environment

This functionality is already intended to be a basic part of Core. Additional levels of support that we can offer in SL_0 are definitions for classes that are either common to military systems, such as weapon systems, and command and control entities, or that are pervasive and problematic to describe, such as terrain features and transportation networks, weather systems, and communications systems.

7.3.2 Describing System Dynamics

Our design of SL_0 is primarily constructed around a time-control routine that takes an event-scheduling approach similar to that found in Simscript. Events are represented by a sequence of state-changing actions and maintained on a time-ordered list. Simulation time is recorded by a clock object and advanced by setting the clock to the time of the next scheduled event. After the clock is advanced, all events scheduled for that time are activated together. Before advancing time again, the timing routine suspends execution and returns control to a processing loop, which cycles through the active controllers and allows them to select and fire rules.

This approach enables an activity-scanning world view by allowing the actions of rules to schedule events. In this way, each controller can be viewed as representing an activity, while the rules of a controller constitute the discrete phases of the activity. Controllers are allowed to cycle through their rules indefinitely, until no more rules can be fired. At this point a low-priority, default rule to resume the timing routine can be selected and fired, thus allowing the simulation to advance to the time of the next event.

The process-orientation world view can be taken by describing events as process instances. In this way, activating an event is like resuming a process, only resumption is correlated with simulated time. Before detaching, such events would reschedule themselves to be resumed at a later time. Alternatively, these process/events could be

associated with a particular controller, which would reschedule them based upon system state.

Conditional Activation of Events. SL_0 would support several time-dependent enhancements to standard Core types, such as the use of rules to provide conditional activation of events. Normally, the actions of a rule can be thought of as occurring instantaneously. A rule has the general form

(rule *label*
  :if *condition*
  :then *statement*$^+$
  {*property expression*}***)**

where each *property/expression* pair represents a controller-dependent attribute of the rule. We can use Core to define a subclass of controllers, called **simcontroller**, that recognize the property **:delay** as indicating the amount of simulation time that must pass before the instantiation of a rule can be fired.

For example, an instantiation of the rule

```
(rule :detect-radar
  :if (< (distance ?rpv ?radar) (sensing-range ?rpv))
  :then (assert #F:(senses ?rpv ?radar))
  :delay 2)
```

will only be fired if the instantiation is still true after a two-unit delay of simulation time. A simcontroller operates by automatically scheduling an event to fire a selected rule instantiation after its delay time and only if the instantiation is still active. Note that a simcontroller also uses *refraction*, which means that the same rule instantiation can only be fired once. Thus, an instantiation of this rule will be created when an rpv comes within sensing range of a radar; this instantiation will stay active until the rpv moves out of sensing range. If the rpv stays within sensing range for two simulation time units, then the fact that the rpv senses the radar will be asserted. This fact could only be reasserted if the rpv left the sensing range and then entered it again.

Time-Dependent Variables. The next task in defining system dynamics is that of routinizing time-dependent variables and functions. The ideas found here have been adapted from Cammarata, Gates, and Rothenberg (1988). We use the term *continuous-valued function* for all time-dependent functions and variables.

We can identify two uses of time dependency in military simulation. One use is characterized by the fact that the value of a continuous function is constantly being monitored (i.e., any change is important); the other is characterized by the fact that a certain value or threshold

is being monitored for. SL_0 would support facilities for automating both uses. We call the first, *cascaded evaluation*, and the second, *delayed evaluation.*

To illustrate the differences between these two uses, consider the task of graphically modeling an airborne vehicle that detects other objects whenever they are within the sensing range of the aircraft. One way to model this system would represent the aircraft and other entities as simulation objects, where the aircraft has a sensing range and all objects have a position. The position of the aircraft and the distance between it and any other object are both continuous-valued functions. The position of the aircraft is monitored continually by the graphics device, while the distance between the aircraft and any other object is only important when it is less than the sensing range.

Because any change to the position of the aircraft must be represented on a graphic device, it is necessary to use cascaded evaluation on position. Cascaded evaluation simulates continuous computation of a continuous-valued function by incremental evaluation in small discrete steps. This can be done by scheduling an event that evaluates the function, passes the new value to monitors that are waiting for it, and then reschedules itself at a later time. In this way, the event can be described as cascading down the event list.

Distance, on the other hand, is not constantly being monitored. Because it is possible to estimate when the distance between the aircraft and any other object will be within a certain range, delayed evaluation can be used for monitoring distances. Before describing the mechanics of delayed evaluation, we should point out that it is closely related to objects that monitor system state, in particular, switches, which we discussed in Section 7.2. For instance, the switch

```
(switch (and (= (sensing-range ?aircraft) ?range)
             (< (distance ?aircraft ?radar) ?range))
 (let ((fact #F:(senses ?aircraft ?radar)))
 (assert fact)
 (detach)
 (retract fact)))
```

monitors the distance of the aircraft from a radar site, asserting the fact that an aircraft senses a radar site when it is in range and retracting that fact when it leaves range. The condition of a switch can be monitored using a variation on the RETE algorithm. A variation that we can use is to recognize that conditions, such as

```
(< (distance ?aircraft ?radar) ?range)
```

require delayed evaluation.

In discussing how delayed evaluation works, the following terminology is used (some familiarity with the RETE algorithm is assumed). The node (NODE) of the RETE net monitoring for a condition requiring delayed evaluation is called an *expectant*. The arguments to the function (ARGS), the threshold value (VAL), and the logical operation indicating whether the function is expected to be less than, equal to, or greater than the threshold (OP) are collectively called an *expectation*. The object that monitors delayed evaluation of a particular function is called an *it evaluator*. Note that the same evaluator monitors all expectations on the same continuous function.

The evaluator requires two functions. One (MAIN) is the function used to compute the value of the continuous-valued function at the current simulation time. The other (ETA) is a function that is given an expectation and returns the time at which the expectation will be true; if never, ETA should fail. The evaluator maintains two lists: one of expectations that will occur (ordered by ETA); and the other of expectations that will not occur (given the current state of the system). The evaluator operates by scheduling the event

(inform *evaluator*)

for the time of the most recent expectation. When this event is activated, the evaluator informs the expectants associated with each expectation estimated to occur at that time that the condition has been met. The expectants can then resume passing information up the RETE net of which they are a part.

There are two other aspects of an evaluator. First, when an expectation is reached, the evaluator must schedule an inverse expectation. (When the inverse expectation is reached, the evaluator must inform the expectant that the condition no longer holds.) Second, an evaluator must be informed when there is a state change that would affect the ETA of an expectation. For instance, if the speed or trajectory of the aircraft changes, then the evaluator will have to recompute the ETA of expectations involving the aircraft and perhaps reschedule the inform event for an earlier time.

7.3.3 Modeling Stochastic Behavior

For reproducing variability and uncertainty, SL_0 would support a pseudo-random number generator, as well as additional features that transform random numbers into variates from standard or user-defined statistical distributions and that perform related sampling tasks.

The pseudo-random number generator would produce a reproducible sequence of real numbers that are statistically independent and

uniformly distributed between 0.0 and 1.0, inclusive. Instances of the class random-state, implemented as streams, are used as seeds for the pseudo-random number generator. SL_0 would also support a class of stream objects called **sample-set** from which statistical samplings can be drawn. A sample-set is created by the form

(make-sample-set
 [**:random-state** *random-state*]
 :function { *function-call* | *distribution-table* })

It consists of a distribution function and related random-state, which can be generated automatically or user-supplied. The distribution function can be selected from a predefined set of standard functions, such as beta, binomial, Erlang, exponential, gamma, normal and log-normal, Poisson, and Weibull, or the user can specify his own function via a distribution table.

When a sampling distribution cannot be readily characterized by one of the standard functions provided, the user can define a distribution function based upon a table look-up mechanism similar to that described in Section 3.3.2. A distribution function so defined is represented by an instance of the class **distribution-table** and created by the form

(make-distribution-table
 :type *type*
 :mode { **:discrete** | **:continuous** }
 :entries
  ((cp_1 s_1)
  (cp_2 s_2)
  . . .
  (1.00 s_n)))

where *cp*, a real number between 0.0 and 1.0, stands for cumulative probability, i.e., $cp_1 = p_1$, where p_1 is the probability of selecting sample value s_1; $cp_2 = p_2 + cp_1$, where p_2 is the probability of selecting sample value s_2; . . . and $1.00 = p_n + cp_{n-1}$, where p_n is the probability of selecting sample value s_n.

7.3.4 Data Collection, Analysis, and Display

Data of interest to a simulation study typically deal with the dynamic aspects of system state. For example, in a simulation of a C^3I system, data of interest might be the average delay in communications caused by different jamming techniques. The collection of such data requires (1) facilities for monitoring system state, and (2) facilities for designating what particular aspects to monitor. The former

requirement is already provided for given the state-based processing facilities of Core; the latter requirement must be provided by SL_0. The user must also be able to specify what to do with particular pieces of data when collected, e.g., collect a mean or collect over a histogram. Finally, the user must be able to specify display formats for data of interest. Such formats could be described in terms of final output or an interactive graphics device.

For this aspect of SL_0 we plan to support a set of declarative statements that allow the analyst to specify which aspects, e.g., attributes, relations, events, etc., to monitor for changes, what to do when changes occur, and how to aggregate and display the accumulated changed data. For efficiency, these statements will be divided into two types, i.e., compile-time and execution-time. The compile-time statements will declare which aspects of a model are likely to be of importance to a simulation study, while execution-time statements will specify exactly which of these aspects are of importance and just what should be done with them. This scheme will allow the compiler to optimize over those aspects that are not going to be monitored.

Explanation and Exploration. Two aspects of data collection, analysis, and display of particular interest to simulation research are those of *explanation* and *exploration*. These topics were briefly touched upon in Section 3, and a somewhat more detailed discussion can be found in Rothenberg (1986).

Explanation requires that the simulation program be able to keep track of what it has done, that it be able to analyze its own event history and dynamic structure, and that it be able to present this analysis in an understandable format. These requirements suggest that explanation is a specialization of more general facilities for data collection, analysis, and display. The specialization lies in automatically determining which aspects of system state to save, how they should be analyzed, and what to show the user. In this vein, the SL_0 will support a minimal explanation facility.

Exploration builds on top of explanation. In general, exploration allows the user to selectively modify a system model, run the simulation ahead, evaluate the changes, and return to a past state. This type of interaction requires that the simulation program be able to save and restore state. For large simulation programs, it would not be possible to make an arbitrary number of core images, so a state-saving facility must make a trade-off between memory space and the complexity of computations required for restoration. As with explanation, SL_0 will support a minimal facility for saving and restoring state; an ancillary facility for helping the user keep track of past states will also be supported.

7.3.5 Advanced Functionality

There are several advanced features suggested by current RAND research that we would also like to include in SL_0. In particular, this would include concepts of *dynamic resolution*, as well as *planning and unplanning*, *constraints*, and *multiple relations*. Below we give a brief overview of these concepts; a discussion of our approach to providing this support is outside the scope of this document. (We would also like to provide support for the concept of decision tables, as found in RAND-ABEL, but, because of their complexity, have not yet made provisions for it.)

Dynamic Resolution. Dynamic resolution refers to the concept of varying the granularity of portions of a system model during the execution of a simulation program. Such a concept is important in dealing with exogenous activities, i.e., activities of the environment in which the system of interest resides. For example, the exogenous activities to a platoon come from a company-level environment. It would be desirable to replicate these activities with a company-level model, and then reduce the granularity of the portions of this model that immediately concern the platoon level (or increase from the platoon back to the company level).

The approach to dynamic resolution suggested by the KBSim project is to actually have two (or more) system models of different granularity and associated *aggregation* and *disaggregation* procedures for mapping between the states of these models. For our example, there might be a company model and a platoon model. The disaggregation procedures would change the state of the platoon model to reflect activities of the company model; likewise, the aggregation procedures would change the state of the company model to reflect activities at the platoon level. Dynamic resolution can also be used to link several fine scale models through the use of a coarser model and to provide analysts with yet another tool for interactive model exploration and design.

Planning, Unplanning, and Constraints. It is often the case in military systems that entities follow a series of activities (a *plan*) in order to achieve some objective. In this sense, the term *planning* refers to techniques for modeling the plans of system entities (as opposed to the AI sense of automatically creating such plans). A plan has a *goal* and a set of *goal constraints*, where the goal represents the objective of the plan and the constraints represent aspects of system state that must be true for the goal to be achieved. For example, a plan whose goal is to take a tactical target would likely have a constraint tied to attrition levels. A plan also has *backup plans*, which are contingencies against failure to achieve a goal due to a certain constraint, and a *stop*

plan, which is a contingency against failure to achieve a goal after exhausting the available backup plans. When an entity switches to a backup or stop plan, it is said to be *unplanning*.

The approach to planning and unplanning suggested by KBSim is to support facilities for representing and using plans and to support a new class of object, called a *planner* object, such that a planner object can use the planning facilities. Work has also been done to integrate these concepts with those of dynamic resolution.

Multiple Relations. In developing their approach to dynamic resolution, KBSim found it useful to define an abstract *part/whole* relation, e.g., a company is modeled as an aggregate entity with parts consisting of a command post and several platoons. Given this relation, KBSim could provide automatic inference support for mapping between the aggregate and disaggregate model. The term *multiple relations* refers to a generalization of this idea, which would allow the definition of other such abstract relations with their own particular inference support mechanisms.

7.4 OVERVIEW OF SLAG

In this section, we give a brief overview of SLAG (Simulation LAnguage Generator). As we have indicated before, SLAG is essentially a compiler-generator for prototype simulation languages. A SLAG program defines a mapping between the surface-level syntax of a prototype simulation language and its System Layer semantics. When designing the syntax of a new prototype, the intention for any such mapping should always be the same: To minimize the distance between the formalization of a system model and its encoding as a simulation program.

The advantages of defining an abstract modeling language on top of some System Layer semantics is that it allows simulation models to be implemented using programming constructs that are intended to seem natural to analysts. As a result,

- Modeling logic merges with programming logic, rather than becoming obscured by it, reducing overhead code
- Errors in the simulation program indicate faults in the model, rather than faults in the model's encoding
- Simulation programs become a means of formalizing and communicating models between analysts.

The abstract modeling language also provides simulation researchers with a platform for demonstrating new ideas and techniques and

comparing them to others. If well thought out, such platforms would be more acceptable to analysts than LISP-based prototype languages, such as ROSS, and would be more likely to be used in developing realistic simulations.

7.4.1 The Role of Syntax

The primary purpose of syntax is to provide a notation for communicating information between a programmer and a programming language processor. However, the choice of particular syntactic structures is constrained only slightly by the necessity to communicate particular items of information, which provides a considerable amount of flexibility in designing the syntax of a language. Thus, the details of syntactic design are largely based on the secondary criteria of *readability*, *writability*, and *expressibility*.

Readability. A program is said to be readable if the underlying structure of the algorithm and data represented by the program is apparent from an inspection of the program text. Readability is enhanced by features that hide the underlying implementation of language semantics from the problem-solving logic of the program. While readability cannot be guaranteed (even the best design can be overcome by poor programming style), poor syntactic design can force even the most well-intentioned programmer to write cryptic programs.

Writability. A language is writable if code generated by a programmer behaves in the manner expected. Where readability is concerned with program comprehension, writability is concerned with the ease by which near "error-free" code can be produced. Writability is enhanced by concise and regular syntactic structures. While the requirements for writability can sometimes be at odds with those for readability, some features, such as abbreviated and resolvably ambiguous constructs, can advance both goals.

Expressibility. Expressibility concerns the ease with which a programmer can encode an algorithm. A language enhances expressibility by supporting syntactic constructs that parallel the vernacular used to formalize the problem. That is, there is a one-to-one mapping from problem-solving concepts to programming constructs. As with writability, syntactic structures that aid expressibility can often aid readability.

To encourage military analysts to use new ideas and techniques, simulation researchers must embed them in a programming language that is acceptable to analysts. As a result, researchers must ensure that this language meets the three criteria outlined above regarding the encoding o. ystem models. The purpose of including SLAG as a part of SERAS is to ease the task of experimenting with the design of such languages, as well as generating them.

7.4.2 Using SLAG

A SLAG program takes the form of an augmented context-free grammar. The grammar defines the syntactic rules of a prototype simulation language (SL_i); the augmentation consists of associated translation rules, which map syntactic structures to their corresponding operational semantic structures. Once defined, a grammar is input to SLAG, which outputs a compiler and interpreter for the target prototype. The interpreter is an interactive programming system intended for use in developing and encoding system models. The compiler takes as input the encoded model (an SL_i program) and generates an executable object file.

SLAG operates by using a grammar to construct parse tables for translating programs into Core programs. These tables, along with a table-driven parse routine, then form a front-end preprocessor to the Core compiler and interpreter and, in this way, give the illusion of generating an SL_i-specific compiler and interpreter.

The design of the parsing component of SLAG has been adapted from a similar system developed in connection with the ROSIE language (Kipps, 1988). This parser is based on a recent algorithm for fast context-free parsing (Tomita, 1985). It requires no reserved words or bounded context restrictions; it handles left- and right-recursion and null productions; and it supports a general mechanism for resolving syntactic ambiguities. Unlike most general context-free parsing algorithms, which take a top-down approach, Tomita's algorithm uses a table-driven, bottom-up approach similar to that found in standard LR parsers (Aho and Ullman, 1972). However, unlike standard LR parsers, Tomita's algorithm emulates a non-deterministic parse, using a form of breadth-first search that merges common subpaths and avoids redundant (and otherwise exponential) computations.

7.5 SUMMARY

In summary, our plan for supporting the transfer of simulation technology calls for the development of a common-base programming system for military simulation, SERAS. The SERAS programming system is targeted for two user groups: simulation researchers and military analysts. It is intended to support the functionality required by analysts within an advanced programming architecture for exploring new modeling techniques.

SERAS provides researchers with a tool for rapidly prototyping new simulation programming languages. New ideas and techniques can be embedded in these languages and made available in this form for

analysts to use and evaluate. Once a promising language has been defined, it can be optimized and used by analysts for realistic applications. Researchers can then continue to explore new simulation techniques and introduce them to analysts as extensions and refinements to this language.

8. FUTURE DIRECTIONS

The focus of our future work would be directed toward the further development of SERAS. This development effort would follow an iterative process of design, implementation, and evaluation. While the initial implementations would be in LISP (for reasons of flexibility), we would move away from LISP as SERAS matures, placing more emphasis on efficiency considerations than flexibility.

It should be obvious from the discussion in Section 7 that SERAS is an ambitious effort. Core is intended to be a new programming language, and not merely a subsystem sitting atop an existing language. Core also integrates concepts from AI and OOP in a new way, so there is no one existing language to use as a model. The baseline prototype language contains features not traditionally found in simulation languages. Because these features are intended to take advantage of the novel programming constructs available in Core, appropriate programming styles and syntax would also have to be developed.

8.1 PHASES OF DEVELOPMENT

As mentioned above, the SERAS development effort would follow an iterative process of design, implementation, and evaluation. Our effort would approximately follow the phases outlined below.

8.1.1 Phase 1

On the first pass through the development cycle, the objective would be to build a demonstrable version of SERAS. This means implementing a minimal subset of Core and the baseline language, as well as a demonstration simulation. The initial implementation would be developed using Common LISP (Steele, 1984) and would borrow from the Common LISP Object System (CLOS) (Bobrow et al., 1988), as instantiated in PCL (Portable Common Loops). At this early stage it should be noted that, because of these two factors, Core would bear a resemblance to both Common LISP and CLOS. While it may always retain these roots, it is not meant to be viewed as an "add-on" to either Common LISP or CLOS. As SERAS matures, it would begin to stand on its own as a programming system separate from Common LISP. (While Common LISP provides a practical environment for prototyping SERAS, there is no current or foreseeable implementation of this

language that can provide acceptable levels of execution-time performance.) By decoupling SERAS from Common LISP, aspects of flexibility can be traded for improved performance.

8.1.2 Phase 2

The objective of the next pass is to refine the design of SERAS and to flesh out those portions of the system already implemented. At this time refinements would be made to the baseline simulation language, and a larger and more realistic system model would be selected for demonstration. The first several implementations of SERAS would include a Core interpreter but not a compiler. As a result, execution time of demonstrations are expected to be slow and should not be used as a metric for evaluating SERAS performance.

8.1.3 Phase 3

Following the development of a large-scale demonstration and resulting refinements to the design of SERAS, work would begin on the implementation of a Core compiler using either C or Ada (but not LISP). Porting SERAS from LISP would cause additional changes to be made to its design and prompt further phases of development.

8.2 ASPECTS OF PROGRAMMER SUPPORT

Upon completing a Core compiler we would begin working on a SERAS programming environment. The objectives of this environment are to support the use of SERAS both as a research tool and a modeling tool. Thus, we foresee that this environment would have two modes of operation: one in which users are programming in Core, and another in which users are programming in a prototype simulation language.

In the first mode, Core would run as an interpreted language rather than a compiled language. It would be very forgiving of programming errors, such as mismatched types, etc., and, in general, provide a lot of flexibility at the cost of execution-time performance. However, for programmers using this mode to develop a prototype simulation language, they must eventually run their code through the Core compiler, which would be much less forgiving and flexible. While this could require a sloppy programmer to spend considerable effort repairing code, the end result, once passed through the compiler, would be a high-performance system.

In the second mode, the programmer would be developing a simulation program within a particular SERAS prototype. While this mode should support tools for easing the modeler's programming task, such tools lay outside the objectives of this project and we have no plans to pursue this type of support. Projects such as the RSAC's RAMP and the KBSim are exploring the design and use of modeling support tools. In recognition of the fact that simulation programs are not always the end product of a system study, once the modeler has finished implementing and testing his simulation, this mode would support facilities for compiling the simulation into an executable file that can take input from external sources and be run as a subprocess to other (non-SERAS) programs.

8.3 CONCLUSION

Without a doubt, SERAS is an ambitious effort. It synthesizes ideas from artificial intelligence, object-oriented programming, and simulation programming into a large and complex programming system. However, SERAS can support the effective transfer of results in knowledge-based and object-oriented simulation by providing a conduit through which these results can be moved into an applications environment. The SERAS approach gives researchers a hook into the language of military analysts, allowing the gentle introduction of new technology. This approach also overcomes the problems of applying a single language to both research and applications, because it separates the language used for defining the semantics of simulation from the language used for encoding simulation programs. SERAS will allow new ideas to be ported from the research environment to an environment that can be used for building realistic models, and, in this way, have a unifying effect on current research and applications in military simulation.

REFERENCES

Abelson, H., and G. J. Sussman, *Structure and Interpretation of Computer Programs*, MIT Press, Cambridge, MA, 1984.

Adam, N. R., and A. Dogramaci (eds.), *Current Issues in Computer Simulation*, Academic Press, New York, NY, 1979.

Aho, A. V., and J. D. Ullman, *The Theory of Parsing, Translation and Compiling*, Prentice-Hall, Englewood Cliffs, NJ, 1972.

Birtwhistle, G. M., O. J. Dahl, B. Myhrhaug, and K. Nygaard, *SIMULA Begin*, Studentlitteratur, Oslo, Norway, 1973.

Blum, J. B., and J. I. Rosenblatt, *Probability and Statistics*, Saunders, Philadelphia, PA, 1972.

Bobrow, D. G., "If Prolog Is the Answer, What Is the Question?" *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 11, 1985, pp. 1401–1408.

Bobrow, D. G., and M. Stefik, *The LOOPS Manual*, Intelligent Systems Laboratory, Xerox Corporation, 1983.

Bobrow, D. G., K. Kahn, G. Kiczales, L. Masinter, M. Stefik, and F. Zdybel, "CommonLoops: Merging LISP and Object-Oriented Programming," *Proceedings of OOPSLA '86, SIGPLAN Notices*, Vol. 21, No. 11, 1986, pp. 17–29.

Bobrow, D. G., L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, and D. A. Moon, *Common LISP Object System*, proposal to the Common LISP Committee X3J13, 1988.

Bratley, P., B. L. Fox, and L. E. Schrage, *A Guide to Simulation*, Springer-Verlag, New York, NY, 1983.

Brownston, L., F. Farrell, E. Kant, and N. Martin, *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming*, Addison-Wesley, Reading, MA, 1985.

Buchanan, B. G., and E. H. Shortliffe (eds.), *Rule-Based Expert Systems*, Addison-Wesley, Reading, MA, 1984.

CACI, Inc., *Simscript II.5 Reference Handbook*, Los Angeles, CA, 1976a.

CACI, Inc., *Simulating with Processes and Resources in Simscript II.5*, Los Angeles, CA, 1976b.

Cammarata, S. J., B. L. Gates, and J. Rothenberg, *Dependencies, Demons, and Graphical Interfaces in the ROSS Language*, The RAND Corporation, N-2589-DARPA, March 1988.

Campbell, J. A. (ed.), *Implementations of PROLOG*, Halstead Press, New York, NY, 1984.

Carnegie Group, Inc., *Knowledge Craft CRL Technical Manual*, Pittsburgh, PA, 1986.

Chao, L. L., *Statistics: Methods and Analysis*, McGraw-Hill, New York, NY, 1969.

Colmerauer, A., H. Kanoui, and M. Van Caneghem, "PROLOG: Theoretical Principles and Current Trends," *Technology Science Information*, Vol. 2, No. 4, 1983.

Conte, S. D., and C. de Boor, *Elementary Numerical Analysis*, McGraw-Hill, New York, NY, 1980.

Conway, R. W., *Some Tactical Problems in Simulation Method*, The RAND Corporation, RM-3244-PR, October 1962.

Dahl, O. J., and K. Nygaard, "SIMULA—An ALGOL-Based Simulation Language," *Communications of the ACM*, Vol. 9, No. 9, 1966.

Dahl, O. J., B. Myhrhaug, and K. Nygaard, *Common Base Language*, Publication No. S-22, Norwegian Computing Center, Oslo, Norway, 1970.

Davis, P. K., and J. A. Winnefeld, *The RAND Strategy Assessment Center: An Overview and Interim Conclusions about Utility and Development Options*, The RAND Corporation, R-2945-DNA, March 1983.

Davis, P. K., S. C. Bankes, and J. P. Kahan, *A New Methodology for Modeling National Command Level Decisionmaking in War Games and Simulations*, The RAND Corporation, R-3290-NA, July 1986.

Fishman, G. S., *Principles of Discrete Event Simulation*, John Wiley, New York, NY, 1978.

Fishman, G. S., and P. J. Kiviat, *Digital Computer Simulation: Statistical Considerations*, The RAND Corporation, RM-5387-PR, November 1967.

Forgy, C. L., "RETE: A Fast Algorithm for the Many Pattern/Many Object Pattern Matching Problem," *Artificial Intelligence*, Vol. 19, 1982, pp. 17–37.

Forrester, J. W., *Industrial Dynamics*, MIT Press, Cambridge, MA, 1961.

Franta, W. R., *The Process View of Simulation*, North Holland, New York, NY, 1977.

Ginsberg, A. S., H. M. Markowitz, and P. M. Oldfather, *Programming by Questionnaire*, The RAND Corporation, RM-4460-PR, April 1965.

Goldberg, A., and D. Robson, *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, Reading, MA, 1983.

Hailpern, B., "Multiparadigm Languages," *IEEE Software*, January 1986, pp. 6–9.

Hayes-Roth, F., D. A. Waterman, and D. B. Lenat (eds.), *Building Expert Systems*, Addison-Wesley, Reading, MA, 1983.

Hewitt, C., "Viewing Control Structures as Patterns of Message Passing," *Artificial Intelligence*, Vol. 8, 1977, pp. 323–364.

Hilton, M. L., *ERIC: An Object-Oriented Simulation Language*, Rome Air Development Center, RADC-TR-87-103, Griffiss Air Force Base, NY, 1987.

Inference Corporation, *ART Reference Manual*, Los Angeles, CA, 1986.

IntelliCorp, *KEE Software Development System User's Manual*, Mountain View, CA, 1985a.

IntelliCorp, *The SimKit System Knowledge-Based Simulation Tools in KEE*, Mountain View, CA, 1985b.

Jefferson, D. R., and H. A. Sowizral, *Fast Concurrent Simulation Using the Time Warp Mechanism, Part I: Local Control*, The RAND Corporation, N-1906-AF, December 1982.

Kahn, K. M., *Director Guide*, AI Memo 482B, Artificial Intelligence Laboratory, MIT, Cambridge, MA, 1979.

Kamins, M., *Two Notes on the Lognormal Distribution*, The RAND Corporation, RM-3781-PR, August 1963.

Kipps, J. R., *A Table-Driven Approach to Fast Context-Free Parsing*, The RAND Corporation, N-2841-DARPA, December 1988.

Kipps, J. R., B. Florman, and H. A. Sowizral, *The New ROSIE Reference Manual and User's Guide*, The RAND Corporation, R-3448-DARPA/RC, June 1987.

Kiviat, P. J., "Simulation Language Report Generators (or, I Hear You But I Don't Know What You're Saying)," The RAND Corporation, P-3349, April 1966; prepared for the Symposium on Simulation Techniques and Languages, Brunel College, London, England, 1966.

Kiviat, P. J., *Digital Computer Simulation: Modeling Concepts*, The RAND Corporation, RM-5378-PR, August 1967.

Kiviat, P. J., *Digital Computer Simulation: Computer Programming Languages*, The RAND Corporation, RM-5883-PR, January 1969.

Kiviat, P. J., R. Villanueva, and H. M. Markowitz, *The Simscript II Programming Language*, The RAND Corporation, R-460-PR, October 1968 (also published by Prentice-Hall, Englewood Cliffs, NJ, 1969).

Klahr, D., P. W. Langley, and R. Neches (eds.), *Production System Models of Learning and Development*, MIT Press, Cambridge, MA, 1987.

Klahr, P., and W. S. Faught, "Knowledge-Based Simulation," *Proceedings of the First Annual National Conference on Artificial Intelligence*, American Association for Artificial Intelligence, Stanford University, Stanford, CA, August 18–21, 1980.

Klahr, P., D. McArthur, and S. Narain, "SWIRL: An Object-Oriented Air Battle Simulator," *Proceedings of the Second Annual Conference on Artificial Intelligence*, American Association for Artificial Intelligence, Carnegie-Mellon and the University of Pittsburgh, Pittsburgh, PA, 1982.

Klahr, P., J. W. Ellis, Jr., W. D. Giarla, S. Narain, E. M. Cesar, Jr., and S. R. Turner, *TWIRL: Tactical Warfare in the ROSS Language*, The RAND Corporation, R-3158-AF, September 1984.

Kowalski, R., "Algorithm = Logic + Control," *Communications of the ACM*, Vol. 22, No. 7, 1979.

Law, A. M., and W. D. Kelton, *Simulation Modeling and Analysis*, McGraw-Hill, New York, NY, 1982.

Markowitz, H. M., B. Hausner, and H. W. Karr, *Simscript: A Simulation Programming Language*, The RAND Corporation, RM-3310-PR, 1962 (also Prentice-Hall, Englewood Cliffs, NJ, 1963).

Maryanski, F., *Digital Computer Simulation*, Hayden Book Company, Rochelle Park, NJ, 1980.

McArthur, D., P. Klahr, and S. Narain, *The ROSS Language Manual*, The RAND Corporation, N-1854-1-AF, September 1985.

Minsky, M., "A Framework for Representing Knowledge," in P. H. Winston (ed.), *The Psychology of Computer Vision*, McGraw-Hill, New York, NY, 1975.

Newell, A., *Studies in Problem Solving: Subject 3 on the Cryptarithmetic Task, Donald + Gerald = Robert*, Technical Report, Center for the Study of Information Processing, Carnegie Institute of Technology (now Carnegie-Mellon University), Pittsburgh, PA, 1967.

Nilsson, N. J., *Principles of Artificial Intelligence*, Tioga Publishing Co., Palo Alto, CA, 1980.

Oren, T. I., "Simulation—As It Has Been, Is, and Should Be," *Simulation*, Vol. 29, No. 5, [illegible], pp. 182–183.

Pascoe, G. A., "Elements of Object-Oriented Programming," *Byte*, August 1986.

Post, E., "Formal Reductions of the General Combinatorial Problem," *American Journal of Mathematics*, Vol. 65, 1943, pp. 197–268.

Pritsker, A.A.B., *The GASP IV Simulation Language*, Wiley, New York, 1974.

Pritsker, A.A.B., and C. D. Pegden, *Introduction to Simulation and SLAM*, Systems Publishing, West Lafayette, IN, 1979.

Radiya, A., and R. G. Sargent, *ROBS: Rules and Objects Based Simulation*, Working Paper 87-001, Simulation Research Group, Syracuse University, Syracuse, NY, 1987.

Reddy, Y. V., and M. S. Fox, "KBS (Knowledge-Based Simulation): An Artificial Intelligence Approach to Flexible Simulation," Technical Report CMU-RI-TR-82-1, The Robotics Institute, Intelligent Systems Laboratory, Carnegie-Mellon University, Pittsburgh, PA, 1982.

Rees, J. A., N. I. Norman, and J. R. Meehan, *The T Manual*, 3rd ed., Computer Science Department, Yale University, New Haven, CT, 1983.

Rich, E., *Artificial Intelligence*, McGraw-Hill, New York, NY, 1983.

Rothenberg, J., "Object-Oriented Simulation: Where Do We Go from Here?" *Proceedings of the 1986 Winter Simulation Conference*, American Statistical Association, Washington, DC, December 8–10, 1986.

Rothenberg, J., R. Steeb, N. Shapiro, S. Narain, S. Cammarata, B. Gates, B. Florman, C. Hefley, S. Bankes, and I. Kameny, *Knowledge-Based Simulation: An Interim Report*, The RAND Corporation, N-2897-DARPA (forthcoming).

Schmieser, B. W., "Random Deviate Generation: A Survey," *Proceedings of the 1980 Winter Simulation Conference*, American Institute of Industrial Engineers, Orlando, FL, December 3–5, 1980, pp. 79–90.

SCi Software Committee, "The SCi Continuous-Systems Simulation Language," *Simulation*, Vol. 9, 1967, pp. 281–303.

Shapiro, N. Z., H. E. Hall, R. H. Anderson, and M. L. LaCasse, *The RAND-ABEL Programming Language: Reference Manual*, The RAND Corporation, N-2367-NA, October 1985.

Sharpe, W. F., *The Army Deployment Simulator*, The RAND Corporation, RM-4219-ISA (Abridged), March 1965.

Shaw, M.L.G., B. R. Gaines, "A Framework for Knowledge-Based Systems Unifying Expert Systems and Simulation," in P. A. Luker and H. H. Adelsberger (eds.), *Proceedings of the Conference on Intelligent Simulation Environments*, SCS Simulation Series 17, No. 1, San Diego, CA, 1986.

Steeb, R., D. McArthur, S. J. Cammarata, S. Narain, and W. D. Giarla, *Distributed Problem Solving for Air Fleet Control: Framework and Implementation*, The RAND Corporation, N-2139-ARPA, April 1984.

Steele, G. L., *Common LISP: The Language*, Digital Press, Bedford, MA, 1984.

Stefik, M., D. G. Bobrow, "Object-Oriented Programming: Themes and Variations," *AI Magazine*, Vol. 6, No. 4, 1986.

Tomita, M., *An Efficient Context-Free Parsing Algorithm for Natural Languages and Its Applications*, Ph.D. Dissertation, Computer

Science Department, Carnegie-Mellon University, Pittsburgh, PA, 1985.

U.S. Department of Defense (DoD), *Reference Manual for the Ada Language* (ANDI/MIL-STD-1815A), Washington, DC, 1983.

Voosen, B. J., *PLANET: Planned Logistics Analysis and Evaluation Technique*, The RAND Corporation, RM-4950-PR, January 1967.

Waterman, D. A., and F. Hayes-Roth (eds.), *Pattern-Directed Inference Systems*, Academic Press, New York, NY, 1978.

Weinreb, D., and D. Moon, *Flavors: Message Passing in the LISP Machine*, AI Memo 602, Artificial Intelligence Laboratory, MIT, Cambridge, MA, 1980.

Welch, P. D., "The Statistical Analysis of Simulation Results," in S. S. Lavenberg (ed.), *Computer Modeling Handbook*, Academic Press, New York, NY, 1983.

West, J., and A. Mullarney, *Extended Modula-2 for Object-Oriented Simulation: Preliminary Language Design Specification*, CACI, Inc., La Jolla, CA, 1987.

Wilson, J. R., A.A.B. Pritsker, "A Survey of Research on the Simulation Startup Problem," *Simulation*, Vol. 31, 1978, pp. 55–58.

INDEX